2016

Detecting and Fixing Emergent Behaviors in Distributed Software Systems Using a Message Content Independent Method

Hendijani Fard, Fatemeh

Hendijani Fard, F. (2016). Detecting and Fixing Emergent Behaviors in Distributed Software Systems Using a Message Content Independent Method (Unpublished doctoral thesis). University of Calgary, Calgary, AB. doi:10.11575/PRISM/25594

http://hdl.handle.net/11023/3078

doctoral thesis

UNIVERSITY OF CALGARY

Detecting and Fixing Emergent Behaviors in Distributed Software Systems Using a Message Content Independent Method

by

Fatemeh Hendijani Fard

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

JUNE, 2016

Abstract

Distributed software Systems (DSS) and Multiagent Systems (MAS) as a sub-class of DSS can provide efficient and cost effective solutions for a wide range of applications. The distributed functionality and/or control in these systems and the local view of the scenarios of the systems can lead to unexpected behavior during execution time, known as Emergent Behaviors (EB) and Implied Scenarios (IS), which was not evident in the requirements and design phase. The new scenarios that are implied to the system can degrade the quality of service and/or cause irreparable damage. Detecting and fixing EB/IS in the early phases, may save costs of software projects by a factor of 20 to 100. In this thesis, we are investigating a new methodology for modeling and analyzing the behavior of software components/agents in order to certify their behavior in advance. Our research questions are: Q1: Is there any methodology that can detect common EB/IS in DSS/MAS without modeling the internal information/knowledge used in software components/agents? Q2: Is there a general approach that can detect EB/IS without human interference and is fully automated? First, we devised a catalogue of the common EB/IS that can arise in DSS/MAS. One of the main advantages of this catalogue is categorizing the EB/IS based on the reasons of occurrence, which helps in devising specific algorithms to detect each type of EB/IS, and can lead to devising solution repositories. The other contribution of our work is devising new modeling based on state machines and social network analysis. This modeling is a general method and can be implemented fully automated. Also, we devised algorithms for detecting the agents that will not show EB/IS in the system as a pre-processing phase. For classes of EB/IS in the catalogue, the detection methodology is devised and recommendations on how to fix the problem are provided. The results of our work shows that all of the EB/IS in various case studies specified in the literature can be detected with our method. Moreover, a new EB/IS is introduced which only can be detected with our modeling.

Preface

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

Journal Articles and book chapters

1. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Catalogue of Emergent Behaviors in Multi-Agent and Distributed Software Systems. " *Submitted* to IEEE Transactions on Software Engineering, 2016.
2. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "On the Usage of Network Visualization for Multiagent System Verification." In *Online Social Media Analysis and Visualization*, pp. 201-228. Springer International Publishing, 2014.
3. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting Neutral Nodes in a Network of Heterogeneous Agent Based System." In *Social Networks: Analysis and Case Studies*, pp. 41-60. Springer Vienna, 2014.

Conference Papers

1. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "A New Approach for the Detection of Emergent Behaviors and Implied Scenarios in Distributed Software Systems - Extracting Communications from Scenarios." In *International Conference on Agents and Artificial Intelligence (ICAART), 2015 7th International Conference on, Doctoral Symposium*, pp. 15-24. 2015.
2. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detection of Implied Scenarios in Multiagent Systems with Clustering Agents' Communications." In *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*, pp. 237-244. IEEE, 2014.

3. **Fard, Fatemeh Hendijani**. "Detecting and Fixing Emergent Behaviors in Distributed Software Systems Using a Message Content Independent Method." In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 746-749. IEEE, 2013.
4. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting Distributed Software Components That Will Not Cause Emergent Behavior in Asynchronous Communication Style." In *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on*, pp. 201-208. IEEE, 2013.
5. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Visualizing the Network of Software Agents for Verification of Multiagent Systems." In *Advances in Social Networks Analysis and Mining (ASONAM), 2013 IEEE/ACM International Conference on*, pp. 1280-1281. IEEE, 2013.
6. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detection and Verification of a New Type of Emergent Behavior in Multiagent Systems." In *Intelligent Engineering Systems (INES), 2013 IEEE 17th International Conference on*, pp. 125-130. IEEE, 2013.
7. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting a Certain Kind of Emergent Behavior in Multi Agent Systems Applied on MaSE Methodology." In *Electrical and Computer Engineering (CCECE), 2013 26th Annual IEEE Canadian Conference on*, pp. 1-4. IEEE, 2013.
8. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "A Method for Detecting Agents That Will Not Cause Emergent Behavior in Agent Based Systems-A Case Study in Agent Based Auction Systems." In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pp. 185-192. IEEE, 2012.

9. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting Emergent Behavior in Autonomous Distributed Systems with Many Components of the Same Type." In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pp. 1924-1929. IEEE, 2012.
10. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Clustering Social Networks to Remove Neutral Nodes." In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pp. 1289-1294. IEEE Computer Society, 2012.

Acknowledgements

I would like to express my warmest appreciation and gratitude to my supervisor, Dr. Behrouz H. Far. He was always my greatest support in accomplishing this research and in my academic life. I have learned a lot of skills from him during these years which helped me to be successful in academia. His support, technical knowledge and valuable feedback, mentoring, and guidance directed me to finish this research and receive AITF and Killam prestigious awards. I was lucky to be one of his PhD students and always valued our discussions and expressing ideas about various topics. I was always motivated by his support, understanding, and the freedom he gave me in my work.

I am grateful to Dr. Reda Alhajj and Dr. Mahmood Moussavi for providing support and helpful feedback for my work. I would also like to thank the members of my oral defense committee Dr. Diwakar Krishnamurthy, Dr. Mohamed Helaoui, and Dr. Borzoo Bonakdarpour for their time.

I gratefully acknowledge the funding sources Alberta Innovates Technology Futures and Izaak Walton Killam. I was funded two years by each of these sources which made my Ph.D. work possible.

I would like to thank my colleague Mr. Emad Mohammed who provided insightful questions and discussions about my work and helped me in conceptually understanding some mathematical theories. I am also thankful to Dr. Elham Akhoundzadeh for her support in reviewing parts of my work which are related to data mining and giving me valuable information for these sections.

I finally thank my husband, Ehsan, who was always my greatest support throughout these years. I really appreciate his encouragement to acknowledge my abilities and helping me to be confident enough to talk about my ideas. Without his support and patience, I was not able to accomplish this work as it is.

To my parents Hafezeh and Nader
For their care and support in my life
And teaching me how to learn

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ABS	:	Agent Based System
CLEB	:	Component Level Emergent Behavior
DM	:	Data Mining
DSS	:	Distributed Software System
EB	:	Emergent Behavior
FSM	:	Finite State Machine
hMSC	:	High Level Message Sequence Chart
IS	:	Implied Scenario
ISSL	:	Intelligent Software Systems Laboratory
ITU	:	International Telecommunication Union
MAS	:	Multi Agent System
MPI	:	Message Passing Interface
MSC	:	Message Sequence Chart
MSG	:	Message Sequence Graph
SD	:	Sequence Diagram
SLIS	:	System Level Implied Scenario
SN	:	Social Network
SNA	:	Social Network Analysis
UML	:	Unified Modeling Language

Chapter One: **Introduction**

In this thesis we provide a new methodology to model and detect the unexpected behaviors that occur during runtime in Distributed Software Systems (DSS) and Multiagent Systems (MAS – a sub-class of DSS) in order to make the whole process fully automated without requiring the expertise of a human.

DSS and MAS are a class of software that run on physically distributed computing devices in which functionality and/or control is distributed. DSS and MAS can provide efficient and cost effective solutions and services for commercial applications such as manufacturing systems and information management and retrieval.

In software engineering, a practical approach for development of Distributed Software Systems is describing system requirements using scenarios (scenario based specification). A scenario is a temporal sequence of events among system components that describe how system components interact to provide functionality of system. UML Sequence Diagrams (SD) [1] or ITU-T Message Sequence Charts (MSC) [2, 3] and high level Message Sequence Charts (hMSC) are examples of artefacts used for describing system functionalities with scenarios.

Lack of centralized control and multiplicity of scenarios in DSS and MAS imply that the quality of service may degrade for collaborative tasks because of possible but unwanted behavior at the run time, commonly known as Emergent Behavior (EB) at the component level (i.e. analyzing the behavior from the local view point of a single component) and Implied Scenario (IS) at the system level (i.e. behavioral analysis where more than one component is investigated) [4]. EB/IS is “unpredictable” behavior that was not evident to the designer during the requirements and design phase of the system. Unpredictable systems are hard to debug and harder to manage and may cause critical and irreparable damages.

Fault in software (runtime) and incorrect software design has resulted in many system failures and has shown many hazard or costly examples both to human life and industry [5-7]. It is stated that "*up to 70% of all faults detected in large-scale software projects are introduced in requirements and design*" [8]. Some researches study human factors that influence the quality of software design or refer to the attributes of the design reviewers that affect on software [7].

Therefore, software needs to be verified in order to assure that it meets the requirements and specifications [5]. Certifying the behavior of DSS and MAS guarantees the reliability and quality of software. Several guidelines and standards are developed which explain the required steps in each phase of software lifecycle. However, the manual reviews are not effective for reviewing and detecting the flaws in software requirement and design and automatic approaches are more appreciated. Moreover, the cost of detecting faults in the early phases of software lifecycle is much more less than detecting them in later phases and after its implementation [9]. Because if the fault exists long time in a software lifecycle, it is more costly to detect and remove it, and it is not guaranteed to be corrected properly. "*Detecting the causes of faults early may reduce their resulting costs by a factor of 100 or more*" [8]. Thus, the cost of detecting and removing errors increases in the late steps of development [10]. Also in industrial cases, every hour of inspection in requirement phase of software has 30 times return on investment, or 33 hours of maintenance is saved for each hour dedicated in requirement phase [11]. Therefore, certifying the behavior of software in the requirement and design phase has a great advantage in terms of saving costs, time and effort. The analysis of DSS from its requirements and design documents has a great attention in the literature.

There are three approaches to manage EB/IS in different phases of the software life cycle: (1) run-time analysis (i.e. executing the system and analyzing the logs); (2) static analysis (i.e.

walk through the software and tracing behavior of its components); and (3) model-based analysis (i.e. using design documents to synthesize the behavior of individual components and the system).

Model checking in this context examines the software behavior (explained in a structure or specification language) against a model which shows the behavior of software. If these two are equivalent then software is promised to execute the exact functionality as it was specified in its requirements. If the model does not satisfy the specification language, it is said that the components may show an unexpected behavior, namely EB or IS [4].

The model checking approaches have specific applications such as network analysis in which the bit values of the properties are checked in each model, especially if the safety properties of the system are defined. Among various applications, the model checking approach that is used for the detection of EB/IS in DSS is the main research area in the background study in this thesis. Since scenario based specifications has taken a lot of attention especially in DSS [2], a lot of researches are dedicated to investigate errors in scenario based specifications [12, 13]. We also developed a new methodology for the detection of EB/IS in scenario based MAS and DSS, which is based on model checking and social network analysis.

Model checking has lots of advantages, but industrial cases for DSS and MAS may face some challenges using them. In the researches working in this area, there are some common problems that we mention in the following:

1. (P1) The process of constructing behavioral models is complex and hardly scalable.
2. (P2) The methods are message content dependent and require domain expert intervention, which makes it hard to be fully automated, since it depends on the application.
3. (P3) Differentiating between send and receive messages and synchronous and asynchronous communication is ignored in many works.

4. (P4) The interactions of software components are not considered and not analyzed.
5. (P5) The interaction among software components is not considered, which makes it hard to investigate the interaction problems and system level analysis.
6. (P6) They lead to state space explosion problem and only recently some distributed methods are developed to overcome some parts of scalability problem.
7. (P7) EB/IS can still exist between some components of the same type.
8. (P8) They mainly focus on detection rather than providing suggestions for fixing the problem.
9. (P9) Behavioral model should be constructed again when requirements are added/changed.
10. (P10) Modeling the properties to check requires expertise with the modeling language.

Referring to these issues, we have developed a new methodology for the detection of EB/IS in MAS and DSS. We approach this problem by a different modeling and classifying the common EB/IS based on the reasons that can cause EB/IS in the execution time. The specific motivation of this work is providing answer to the following questions: Q1: Is there any methodology that can detect common EB/IS in DSS/MAS without modeling the internal information or knowledge used in software components/agents? Q2: Is there a general approach that can detect EB/IS without human interference and is fully automated? Q3: Is there an approach that considers the interactions among software agents in order to certify their behavior in early phases?

In this chapter, we explain the achievements and contributions of our work starting by the explanation of motivations of doing this research, followed by the research goals, and our methodology. We have devised this methodology to overcome some of the above mentioned problems, specifically in the behavioral modeling. In summary, this methodology has four main phases:

1. Phase I: Transferring the scenarios into analyzable data structure and modeling the system.
2. Phase II: Pre-processing phase to remove the components with no EB from further analysis.
3. Phase III: Categorisation of the EB and IS that can happen in MAS and DSS.
4. Phase IV: Devising algorithms for some classes of EB/IS for the detection of potential EB/IS in the system.

In the following chapters, the detailed tasks in each phase will be explained. In each chapter, we explain the problem that we focus in each phase, and then we define our methodology and strategy that we have used to accomplish each phase. After explaining these phases, we give details about the advantages and completeness of our approach with chapters on case studies and results. In these chapters, we have discussed the differences of our work with other works and that our methodology can detect the implied scenarios of other works, as well as new implied scenarios that cannot be detected with other approaches. For detecting each class of EB/IS, we have defined general algorithms. In each of these algorithms, various functions can be used for separate parts. For example, different algorithms might be applied to find the shared states or shared interactions of each or a set of components. Various algorithms for this process is one of the future trends of our research. We have used data mining and clustering in parts of our methodology to detect shared states.

It is worth mentioning that in this thesis, we may use the words component, process, and agent and by all of them we mean the software components that are used in DSS.

1.1 Motivation

1.1.1 Problem definition

The verification of DSS has taken a special attention due to the growing demand of having DSS in this decade. DSS are a class of software in which functionality or control is distributed. This may cause the components of DSS to emerge an unexpected behavior which was not seen in the requirements or designs of the system. Deadlock, race conditions, process divergence, and implied scenarios are kinds of unexpected behaviors [9, 14-18]. This unpredictable behavior known as emergent behavior can have irreparable damages [9, 19] with lots of cost to fix it [4]. The cost of detecting and fixing emergent behavior in requirement and design phase is about 20 times cheaper compared to fixing it after deployment [4].

Software V&V in requirement phase is in the form of making sure that the user needs are satisfied [10, 20]. Formal software V&V in requirement is referred to as model checking which is used to detect race conditions, deadlocks, and other incorrect or unexpected behaviors that may occur [21-23]. Some researches refer to formal methods as mathematical methods to check all the state space of software components and modules and prove the correctness of designs or detect behavioral anomalies [24, 25]. Model checkers use exhaustive coverage (i.e. checking all of states in the state space) and require building finite state or automata models [26-28].

One of the issues about using formal methods is their performance. Many model-checker tools provided for academic researches do not have good performance or are not user friendly [29]. The other main issue is their scalability. Model checking approaches are not scalable by large systems with many components, because the state space expands exponentially by adding any task or component to the system and causes the problem of state space explosion [5, 20, 22, 29-33]. This becomes even worse in DSS and agent based systems, where the state space grows extremely

fast with the number of states in processes. Also, for DSS and Mas, the types of errors in these systems cannot be easily detected by inputting sample data, since they occur in sending and receiving data by the processes at special times or in certain sequence [25]. There are some efforts to overcome this issue, e.g. using flat automata, but they may affect the quality of model checkers by not finding all the errors they are promised to detect.

One of the other problems with most of formal method languages, behavioral modeling and model checkers is their cost, the amount of time for defining the constraints and rules, besides expertise and knowledge required for the applications and language notations or techniques [5, 34-36]. The complexity of formal methods and misunderstanding of engineers using them may cause the formal method to be useless instead of taking its advantages [8]. Also, in the requirement model checking of scenarios, these approaches cannot identify the exact location of the scenario specification causing errors [37]. Moreover, in many works, only the existence of a problem is notified and no clue is provided on where and how the problem has occurred. This makes it hard to fix the exact location of the problem or solve it.

1.1.2 Specific problems targeted by this research

The issues stated above motivated us to devise a methodology for detecting requirement and design flaws of DSS automatically. The deficiencies found with existing approaches inspired us to devise a new methodology for emergent behavior detection. The detection in early phases of requirement is focused to make the certification of software development cost effective.

Specifically talking about researches working on requirements of the system, the following deficiencies are found and focused in this research:

Problem P1: The process of constructing behavioral models is complex and hardly scalable. The researches that investigate the system requirements and designs for detecting emergent behavior mostly use the automaton theory and modeling the behavior of components of the systems [38]. Then by applying various algorithms or defining special languages they detect emergent behavior of the components of the system. Therefore, it needs special expertise, algorithms, and tools to model components' behavior besides the emergent behavior detection. As mentioned before, this process (behavioral modeling) is time consuming and complex [4].

Problem P2: The existing methods that use behavior modeling are message content dependent. This requires a great time and effort to verify the specifications if system requirements change, e.g. adding a new component or modifying interactions between existing components. In this case, the whole process of behavioral modeling and then detecting emergent behavior should be done from the beginning. Besides, the message dependency in this level requires domain expertise to annotate the model or specify proper specifications [4, 12, 30]. In other words, the methodology is based on the application and is not general. Therefore, for each system, it requires domain expertise, which make the automation process hard.

Problem P3: While system requirements especially in scenario based systems show the interaction of all types of components of the system, they cannot show this interaction among all instances for each type. Therefore, emergent behavior can still exist between some components of the same type (e.g. all sellers in an online auction system are of the Seller type); but existing research do not handle this.

Problem P4: In the process of detecting emergent behavior, differentiating between send and receive messages is not considered in many researches [37], or needs identifying specific definitions to recognize between send and receive messages between the components. While in

real world, send messages are not received at the moment they are sent. This makes a flaw between detecting emergent behaviors in requirement phase and what really happens in system execution.

Problem P5: The existing research, only notifies the existence of a problem and does not provide details on where and how the detected EB/IS can happen. Only in some works, and on specific case studies the explanations are provided, which are not expandable to other application areas. This process, makes it hard for the designers to identify the location of the problem and provide solutions for the detected EB/IS.

1.2 Research goal

The above problems **P1-P5** motivated us to find a new message content independent technique for the detection of EB/IS in requirement phase of scenario based specifications in MAS and DSS. We aim to provide a general methodology (not application specific) that considers the software components as black boxes and tries to find the reasons of EB/IS in a wide range of DSS/MAS.

Research questions: The following research questions will be addressed in this research:

Q1: What are the deficiencies of the existing approaches for emergent behavior detection problem? These are summarized in problems P1-P5.

Q2: What are the specific types of EB/IS related to DSS and MAS?

Q3: How to define a message content independent method (a general method) to detect and resolve those problems?

The proposed methodology to answer the research questions is explained in next section.

Research goals: The main goals of this research are:

Goal G1: Classifying common emergent behaviors and implied scenarios in MAS and DSS.

Goal G2: Devising a message content independent technique and tool for the detection of a subset of classified emergent behaviors of G1 in DSS addressing the P1-P5 problems.

1.3 Methodology

Based on the problems defined in the previous section for different approaches to detect emergent behavior in scenario based software, devising a message content independent method is considered as a valuable technique in our work. The importance of this technique is providing a general methodology which can be implemented fully automatic without the intervention of domain experts to define the system properties for certifying the behavior of software components. This technique is inspired by social network analysis in which a goal is to find nodes that violate regular interactions or to investigate nodes' communications to find a new path or link in their relationship [39]. This is similar to the detection of unexpected behaviors in DSS. In DSS there are several components without a central controller which are interacting with each other. The interaction of DSS components in the scenarios in the form of a MSC or high level MSC (hMSC) makes a graph like structure. This graph represents the components' interactions. In the MSCs and hMSCs, an emergent behavior can arise in the form of a new interaction or a new scenario, which is interpreted as a new path or extracted graph compared to the basic ones. Therefore, although considering that each component behaves exactly as its specification, the interaction of various components in different scenarios of the system may emerge a new behavior. In this methodology, our technique uses concepts from state machines and social network analysis. We consider the messages sent/received to/from each agent as the states and consider it as the nodes for each agent. We also consider the corresponding state on the other agent as another node in the graph, which represents and preserves the interaction information for each agent. Therefore, the extracted graph can be

mapped to what is visually shown on the scenarios. The reasons of emerging a new behavior in the run time are then applied to the extracted graphs to investigate if EB/IS can exist. The detected EB/IS are the violations found in the model with regards to the first modeling that represent the components' behaviors as it is seen visually in the scenarios of the system. For the detection process, we extract specific vectors from the graphs. This process helps in identifying the exact cause of the EB/IS. Moreover, based on these reasons, we develop solution repositories in order to prevent the detected type of EB/IS. This will guide the designer to better revision of the specifications.

1.3.1 Tasks

The focus in this research is on automating the process of modeling, analyzing, detecting, and resolving the emergent behavior among DSS components. The detailed tasks are:

1. **Task T1:** Devising a message content independent technique to transfer the scenarios to analyzable data structures. The technique uses graph definition in the form of Core and Node to investigate the component and system level EB/IS separately, and only model the system once. Related definitions and more details are found Chapter Three.
2. **Task T2:** Applying a pre-processing phase to remove components with no EB from further analysis in the component level. We describe the process in Chapter Four.
3. **Task T3:** Classification of common emergent behavior types and using them as marker. We use this classification as a marker for detecting emergent behaviors in our technique. Furthermore, this classification is required for fixing the detected emergent behaviors. One of the examples of this classification in agent based systems is the problem arising when

one agent is missing the information about the senders of same messages. This is one of the categories defined through this research. We explain this catalogue in Chapter Five.

4. **Task T4:** Developing a solution repository for each category of emergent behavior to fix them, either in the design of the components or in composition of scenarios. This is explained in Chapter Six.
5. **Task T5:** Developing real-world case studies for various sectors and from the literature which is explained in Chapter Seven and partly in Chapter Eight.
6. **Task T6:** Developing a tool to provide support for this process. We explain our tool named Eagle in Chapter Eight.

Tasks **T3** provides specific process to achieve goal **G1**. Goal **G2** is accomplished through the other tasks.

1.3.2 Evaluation

The evaluation of finding EB/IS is a challenging area, since there is no specific benchmark or data set for the methodologies. There are some researches which are considered as the main works in this field. In these works, the evaluation is accomplished by applying the methodology on various case studies. For this thesis, we will do evaluation by a combination of all the evaluation methods used in other works. We will evaluate the results by comparing it to the work of Song el at. (2009-2011) [37], Uchitel et al. work (2003) [40], Kumar work (2010) [41], and similar works; and by applying our methodology to all the case studies used in the main researches in this field. The above mentioned groups have done a combination of theoretical and practical work, or their work includes critics on the flaws of the other ones. Therefore, it provides a wide range of criteria for comparison of our methodology to the others. For example, Song uses *unenforceable graphs* to

detect implied scenarios and his method works for different communication styles. Their work in among the only works that approach the problem in a way other than state machines. Uchitel on the other hand provides a tool for his work and detects implied scenarios in hMSCs. He uses lots of case studies from literature as the main evaluation of his work. Kumar claims to propose a correct and complete method which also critics other works such as Uchitel and Muccini. His work is published on 2010 and uses Message Sequence Graphs which is like hMSC. Therefore, the comparison with these works covers almost all evaluation features. The results of our evaluation and discussions are provided in Chapter Eight and Chapter Nine. In our evaluations, we have also assessed the completeness of our methodology by checking the implied scenarios it can detect against the other methodologies in the literature.

1.4 Contributions

The following list represents the contributions of our work:

1. Certifying the behavior of DSS and MAS from the MSC/SD by considering them as black boxes. This leads to have a method which is:
    a. Message content independent.
    b. General and not being application specific.
    c. Has the ability to be fully automated, since it does not require human intervention.
    d. Does not require the internal states or knowledge space of the software components/agents.
2. Classifying the common types of EB/IS in MAS/DSS which we refer to as EB Catalogue. This also has led to:
    a. Introduce a new emergent behavior that is caused by considering the interactions among agents in the behavioral modeling.

b. Finding the origins of the problem.
c. Being able to suggest solutions to fix or prevent the problem.

3. Adopt behavioral modeling using a combination of the concepts from automaton theory and social network analysis, and using the interactions among components:
   a. Preserve the agents' interactions in the model.
   b. Transform it to send and receive interaction matrices and analyze various communication styles.
   c. No need to model the behavior of the system again, when requirements are added/modified. In this case, only the added/modified scenario should be modeled.
4. Adding a pre-processing phase to optimize the component level analysis by detecting components that have no emergent behavior.

There is another research trend on the detection of implied scenarios in the Intelligent Software Systems (ISS) Laboratory under the supervision of Dr. B. H. Far. This trend [12, 42] is mostly based on Mousavi's work on the detection of implied scenarios using the causality order between messages [4]. These works are expanded to the detection of implied scenario in MAS, using AUML methodology, and enhancing the construction of the causality table by using ontology and finding related architectures for the system. The difference of this work with these researches is the usage of a new methodology for modeling the behaviors of components and detecting the issues in component and system level. As mentioned previously, in this work we try to look at the system as black boxes and try to find a solution which is more general and can be fully automated.

1.5 Publications

The progress of this research has been presented in a number of publications as follows:

Journal Articles and book chapters

4. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Catalogue of Emergent Behaviors in Multi-Agent and Distributed Software Systems. " *Submitted* to IEEE Transactions on Software Engineering, 2016.
5. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "On the Usage of Network Visualization for Multiagent System Verification." In *Online Social Media Analysis and Visualization*, pp. 201-228. Springer International Publishing, 2014.
6. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting Neutral Nodes in a Network of Heterogeneous Agent Based System." In *Social Networks: Analysis and Case Studies*, pp. 41-60. Springer Vienna, 2014.

Conference Papers

11. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "A New Approach for the Detection of Emergent Behaviors and Implied Scenarios in Distributed Software Systems - Extracting Communications from Scenarios." In *International Conference on Agents and Artificial Intelligence (ICAART), 2015 7th International Conference on, Doctoral Symposium*, pp. 15-24. 2015.
12. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detection of Implied Scenarios in Multiagent Systems with Clustering Agents' Communications." In *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*, pp. 237-244. IEEE, 2014.
13. **Fard, Fatemeh Hendijani**. "Detecting and Fixing Emergent Behaviors in Distributed Software Systems Using a Message Content Independent Method." In *Automated Software*

Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp. 746-749. IEEE, 2013.

14. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting Distributed Software Components That Will Not Cause Emergent Behavior in Asynchronous Communication Style." In *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on*, pp. 201-208. IEEE, 2013.
15. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Visualizing the Network of Software Agents for Verification of Multiagent Systems." In *Advances in Social Networks Analysis and Mining (ASONAM), 2013 IEEE/ACM International Conference on*, pp. 1280-1281. IEEE, 2013.
16. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detection and Verification of a New Type of Emergent Behavior in Multiagent Systems." In *Intelligent Engineering Systems (INES), 2013 IEEE 17th International Conference on*, pp. 125-130. IEEE, 2013.
17. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting a Certain Kind of Emergent Behavior in Multi Agent Systems Applied on MaSE Methodology." In *Electrical and Computer Engineering (CCECE), 2013 26th Annual IEEE Canadian Conference on*, pp. 1-4. IEEE, 2013.
18. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "A Method for Detecting Agents That Will Not Cause Emergent Behavior in Agent Based Systems-A Case Study in Agent Based Auction Systems." In *Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on*, pp. 185-192. IEEE, 2012.
19. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Detecting Emergent Behavior in Autonomous Distributed Systems with Many Components of the Same Type." In *Systems,*

Man, and Cybernetics (SMC), 2012 IEEE International Conference on, pp. 1924-1929. IEEE, 2012.

20. **Fard, Fatemeh Hendijani**, and Behrouz H. Far. "Clustering Social Networks to Remove Neutral Nodes." In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pp. 1289-1294. IEEE Computer Society, 2012.

1.6 Awards

During the course of this thesis, we were honored to receive the following awards:

1. 2014, 2015 Izaak Walton Killam Memorial Scholarship
2. 2013, 2014 Alberta Innovates Technology Futures (AITF) Doctoral Scholarship
3. Eyes High Leadership Doctoral Scholarship
4. 2014, 2015 Killam Research Scholarship
5. 2014 ECE Top-up Award
6. University Research Grants Committee (URGC) Graduate Conference Travel grant
7. 2013, 2014, 2015 Research Productivity Award, Department of Electrical and Computer Engineering
8. 2012, 2013, 2014, 2015 Annual Progress Report (APR) Excellence Award

1.7 Summary

This thesis tries to find a new methodology for modeling DSS and MAS in scenario based specification systems in order to detect unexpected behaviors during runtime that we refer to as emergent behaviors (component level) and implied scenarios (system level). The main motivation of our work is considering components/agents as black boxes as well as covering some other

problems of the current approaches. In this chapter, we explained the main motivations, problems targeted by this work, and our methodology. The tasks that we should accomplish to achieve the research goals as well as the evaluations are included. This is followed by mentioning the contributions of our work and the list of publications and awards that we have received for this research.

Chapter Two: **Literature review**

2.1 Introduction

In this chapter the related works on detecting software faults are explained in the first section. The related works starts with introducing formal software verification and validation (V&V), the formal methods, and model checking to detect software errors. The introduction then narrows to a specific kind of software fault or anomaly behaviors of software known as emergent behaviors or implied scenarios. These behaviors are the unexpected behaviors of software components emerged in the execution time which were not seen in the requirement and design of the software system. The detection of errors in the early design saves a lot in software cost, and this research is focusing on detecting such behaviors in the requirement phase. The related works and different approaches of detecting emergent behaviors in early phases of software lifecycle are explained in detail in section 2.3.

Concentrating on the target systems of this research, problems with the existing approaches (see section 1.1), and the intended technique devised for this problem, the last two sections goes through the scenario based specifications with Message Sequence Chart (MSC) in DSS and SNA.

The field of emergent behavior and implied scenario detection in the Multiagent Systems (MAS) should be differentiated from other research topics such as MAS model checking, Runtime verification, and static analysis. In order to specify the differences of this work with the research of this thesis, specific sections of this chapter are dedicated on a brief background study on these fields.

Moreover, the automata based modeling approaches can face some problems in detecting emergent behaviors and implied scenarios in MAS and DSS, which will be explained in this chapter. The details of the following topics are given in this chapter:

1. Software verification and validation
2. Formal methods
3. Model checking
4. Emergent behaviors and various detection approaches
5. Scenario based specification and Message Sequence Charts
6. Social network analysis
7. Model checking Multiagent Systems
8. Runtime verification

2.2 Software verification and validation, formal methods, and model checking

2.2.1 Software verification and validation

Software Verification and Validation has tasks for each phase of software development lifecycle [43, 44]. There are many researches working on software V&V and formal methods [30, 34, 44, 45] to verify the behavior of embedded or hybrid software, distributed software, program codes, concurrent and parallel programs and etc. [22, 26, 35, 46-54]. In some other works, the V&V refers to system requirements in the form of making sure the user needs are satisfied [10, 20].

Many tutorials, guidelines, and standards are published for software V&V. IEEE Std. 1012/1998 is one of main standards for software V&V and is revised in various years [6, 55-58].

2.2.1.1 Formal methods

Formal methods are the mathematical based techniques, tools, and notations for specifying and reasoning software and hardware systems. They are used in different phases of software lifecycle for modeling and verifying requirements, designs, architectures, program code, etc. and for different systems: DSS, service oriented systems, concurrent and real time systems, embedded systems, etc. [8, 28]. Formal methods are categorized into two main groups: Model checking and

Static analysis. The former models the behavior of system with logical expressions and the latter checks the program code without really executing it [24]. JLint, PolySpace, and ESC are examples of tools for static analysis [27].

Formal methods are developed for various purposes for systems [25]. Specification languages are used to express the system behavior in mathematical way. There are many of specification languages developed: Z notation [59], VDM (Vienna Development Method) [60], RAISE (Rigorous Approach to Industrial Software Engineering) [61], ASML (Abstract State Machines Language), CCS (Calculus for Communicating Systems), LOTOS (Language of Temporal Ordering Specifications) as a part of SCAFFOLD (Support for the Construction and Animation of Formal Language Descriptions) [62],SDL (Specification and Design Language) [63]. Some other like state machines e.g. RSM (Recursive State Machines) [64] or automata [65] are used to model certain systems. RMSL and SCR are requirement specification languages developed for error checking in the first steps [10].

There are also visual formalisms like State Chart and OCL (Object Constraint Language) considered as a part of UML for formalizing constraints [66], Petri Nets [67], SDL (Specification and Description Language) [68], ROOM, SA [61], FSM (Finite State Machines) [19], and etc.

Among the specification or modeling languages and diagrams, SDL, CSP (Communicating Sequential Processes), CCS, IOA (Input Output Automata which is a nondeterministic state machine), Z, FSM , State charts, Petri Nets, UML (modified as AUML) are used or can be used for distributed and agent based systems showing the system behavior and structure [25].

The events and states or the transitions are used with different definitions for the system behavior in different phases and in each of the methods mentioned above. In some works states are assigned to values of variables, behavior is the sequence of these states and the transition is the

action changing the states [69] and in some other the definitions of state machine are used [61]. In some works the events can refer to the message passing or have other definitions. For example, Petri Nets are defined as 5-tupple (P, T, I, O, M), where P shows a set of places, T represents set of transitions, I is a set of inputs, O is a set of outputs, and M shows set of initial markings. In Petri Net, the tuple is shown in a directed graph containing places and transitions [34].

2.2.1.1.1 Model checking

Model checking is considered as a category of formal methods that verifies the software system with respect to logical specifications. It is used to detect the behavioral anomalies and software faults [24] and study the modeling of interactions of system components [52].

Model checkers use a structure (e.g. finite state machine) and a logical formula (e.g temporal or propositional logic) to check whether the structure is a model of the given formula [1]. The formulas explain the target properties of the system to be checked. Model checking process for DSS examines all the states and inspects their reachability [5].

In requirement verification, the user's provided specification for the proper behavior of the system is defined and the program's execution is verified against it. One of the techniques in this verification is the use of simple state machines like Finite States Machines (FSM), or its extensions like Timed Automata (considering time constraints on each state) [70] and Omega Automata (accepts infinite states) [19, 43].

However model checking showed successful results in some systems (e.g. SPIN for NASA's DEEP SPACE1 [21, 71], using Z specification for IBM CICS and reduce the development costs, ZING model checker, Behave! [72]) there are some challenges using them. Although exploring all states is an advantage of this approach, makes it hard to scale up and often faces state space

explosion problem. The other challenge is the use of notations and models by engineers for designing large scale systems or lack of enough tool supports [73].

Model checking runs all the states of all the processes in the system in parallel. In general, the states of the processes are connected to each other with e-move. This will be the state space of the whole system. Since the states are connected with e-move, the non-deterministic problem will occur. This leads to have a large state space for the system, which is known as state space explosion.

Some methods and algorithms are proposed for the state space explosion problem like *abstraction* and *predicate abstraction* for verifying the program. These approaches do not search the entire state space and they do not guarantee that a property never will be violated. One other issue mentioned in some researches is that model checkers detect the existence of an error and they do not prove or guarantee that there is no errors [5, 24].

SPIN is one of the most popular model checking tools. SPIN is bounded by the size of available memory and the size of reachable states [74]. In the past years some new verification algorithms using parallelism, multicore, and GPGPU are developed to increase the size of state space under investigation [75, 76]. The specification of the model that the validation should be done against it requires specific knowledge, and become complex as the software becomes more complex. Therefore, new field of research is founded for mining the specification from existing systems [77]. In addition, model checking tools generate counterexamples to represent whether a model has faults. The detection of the roots of this fault in a model requires a significant amount of manual work [78]. Java Pathfinder [79], SLAM [80], Bandera [81], and BLAST [82] are some of the tools developed for model checking source codes in Java or C languages. In general, these tools are unable to handle complex data, structures, and concurrency.

2.3 Emergent behavior and different approaches for emergent behavior detection

Emergent behaviors in some works are referred to as implied scenarios i.e. when integrating all of the scenarios of the system (e.g. in the form of state machine), they may emerge a new scenario or unexpected behavior. This happens because the model is not the precise equivalent of the requirement (scenario) specification [17, 37, 83-85].

Emerging new behaviors in DSS is more probable because there is lack of central control in these systems. This reason causes the components to have a local view of the system. Consequently, they may start with one scenario of the system and continue in another scenario in a shared state [84-86]. This state in some researches is referred to as identical states [4].

The problem of detecting emergent behavior can be classified in two categories. The component level emergent behavior refers to component fault in some researches where the behavior of the implemented component does not satisfy its specified behavior [8]. It is also known as unit verification when refers to program verification [87]. The second class is the detection of emergent behavior in system level which may refer to as architecture or complex verification in different works [8, 87]. The existing approaches consider one or both of these classes to detect emergent behaviors.

One approach for model checking in the requirement is Alur et al. methodology [14]. His works define a detailed explanation of the model checking of Message Sequence Chart (MSC), Message Sequence Graph (MSG), and high-level Message Sequence Chart (hMSC) [9]. (see section 2.4 for more detail of MSC, MSG, and hMSC)

MSC, MSG, and hMSC are widely used for describing design requirements especially in DSS. They can be considered as an early formal model for the system and can be viewed as models that specify system behavior. All of the linearization of this model is then checked against an

automaton which is defined by the message alphabet ∑, words, languages, states, and transitions which show the behavior of each process or component. If the intersection of the automaton (showing unwanted behavior of the system) and the linearization is empty, it shows that the defined model with MSC satisfies the requirements. He also defines the realizability of MSC and MSG which means the implementation should generate exactly the behaviors specified in the graph. The safe realizability has polynomial-time solution for MSC [88].

Alur et al. investigate methods for synchronous or asynchronous models for MSG. However, the interpreted semantics in each, depends on the way of MSC concatenation. The complexity of each of the models is various:

"*Checking safe realizability of a bounded MSC-graph is PSPACE-hard, Checking safe realizability of a bounded MSC-graph is in EXPSPACE, Given an MSC-graph G and an MSC M over k processes, there is an algorithm that decides in O(|G||M|k) time whether M ∈ L(G), Given an MSC-graph G and an MSC M, it is NP-complete to determine if M ∈ L(G), even when G is a complete graph, or when G is an acyclic graph, checking safe realizability for bounded HMSCs is EXPSPACEhard, and is undecidable in the general case*".

Moreover, the asynchronous MSG model checking is undecidable [14]. One other problem with the model checking with FSM is the processes that is requires before modeling: Flattening, remove cycles, etc. [19].

Whittle and Schumann et al. propose a methodology which uses Unified Modeling Language (UML) notations and investigates the conflicts in translation between UML notations. They define a methodology with algorithms in different steps supported by a tool to synthesize statecharts from Sequence Diagrams (SD), class diagrams, and Object Constraint Language (OCL) specifications. Their main focus is on using their methodology for Agent Based Systems (ABS). Their translation

from sequence diagrams to statecharts can be used for agent skeleton and agent behavior modeling. In this translation, they detect conflicts and indentical states in merging the scenarios to have justified merging of scenarios. The synthesis of statecharts is done in four steps: 1- Annotating each SD with state vectors and define pre-post conditions for each state with OCL. Then conflicts with respect to the OCL spec are detected. 2- Converting annotated SD into flat statecharts, one for each class in the SD. 3- Merging statecharts for each class, derived from different SDs, into a single statechart for each class. 4- Introducing hierarchy to enhance readability of the synthesized statecharts [89, 90]. Their most work is on the design part or synthesizing scenarios to state machines and code [91, 92].

Ben-Abdallah explains the non-local branching choice which causes the emergent behavior in system. This problem may arise from the abstraction view of MSC and not having semantics or specific interpretation of the implemented process when MSCs face a branching choice in hMSC. There is a support tool for analyzing MSC and hMSC named MESA. This tool checks non-local choice, process divergence, and timing consistency for branching and iterating MSC [15, 93].

Considering the non-local branching choice, Muccini inspects its effect on emerging implied scenarios and explains the reason of an implied scenario generation. Implied scenarios can be stated as a new or unexpected behavior of the system which was not seen in the scenarios of the system. When the state machines are synthesized from scenarios (which explain a model and specification of the system), a set of behaviors are presented by state machines which was not in the scenarios [17]. In the non-local choices, different processes send first events in different branches. This is detected by “augmented behavior” of processes in the non-local choice in their algorithm. When processes share the same augmented behavior in the non-local choice, their interaction generates an implied scenario.

Song et al. methodology is quite different with the other approaches. They explain the detection of implied scenarios when using UML as the specification language. Then generate two graphs named specification and implementation graphs. By matching these two graphs, the implied scenarios and the exact cause of its emergence in the specification are identified. Their method considers both synchronous and asynchronous communications [37, 94].

Uchitel et al. provide a method and tool for detection of implied scenarios. Their algorithm builds Labeled Transition Systems (LTS) for behavioral modeling of MSC and hMSC (or as a semantics for MSC) and in the synchronous communication. They have implemented their work in a tool named Labelled Transition System Analyzer (LTSA) [16, 40, 95-97]. The work is also extended by Letier et al. to detect input-output implied scenarios. These scenarios cannot be detected by other approaches developed for implied scenarios [98]. They also refer to the rejected implied scenarios by the stakeholders as negative scenarios and define behavioral constraints with LTS for eliciting them from implied scenarios. It is worth noting that not all implied scenarios are unwanted. The word implied scenario refers to the unexpected behaviors that are not the precise equivalent of the specifications [84]. Kramer et al. use Timed automata and the behavior model specified by LTS to animate the behavior models in LTSA. The Timed automata added local clocks to LTS [99].

Kumar et al. discuss the problems with main researches in this field (Alur, Uchitel, and Muccini). He discusses that many researches are not implementable and amendable, or do not show correctness. They use Message Sequence Graphs (MSG) and FSM and define a reduced transition system to detect implied scenarios and model checking. They develop a complete method for this problem claiming the correctness and ability of implementation of their work [41]. Also they discuss that “without message labels, if we observe one process at a time, checking for

implied scenarios is decidable" [86]. There are other works regarding the theories of message passing automata and regular MSC [100, 101].

A comparison of the main researches explained here can be found in Table 1. The criteria considered for these works are the specification language and modeling they use, the method, supporting of communication styles, complexity, supporting tool, etc.

Table 1: Comparison of different approaches for detection of emergent behavior

Criteria/Researcher	Ben-Abdallah	Alur	Whittle	Uchitel	Muccini	Song	Kumar
year	1998	2000	2001	2003	2003	2010	2000-2010
Specification language or Behavioral modeling (property specification)	MSG LTS (GSTG)	Automata Temporal logic formulas	Statechart	LTS	MFG LTS	Causal graphs	FSM
Scenario specification or Structural analysis (modeling)	MSC hMSC	MSC MSG hMSC	UML SD OCL	MSC hMSC	MSC hMSC graph	UML	MSG
Method for defining implied scenarios	Model checking with Non-local branching choices	Model checking (realizability of state machine)	Model checking with FSM	Model checking (specifications against properties in LTS, MTF & FLTL)	Model checking Non-local branching choices	Graph comparison	Model checking with FSM
Support synchronous (Synch), asynchronous (Asynch)	Checks properties without constraints	Synch. Asynch.	-	Synch.	-	Synch. Asynch.	-
Detect causes of implied scenario	✓	X	X	X	✓	✓	X

Criteria/Researcher	Ben-Abdallah	Alur	Whittle	Uchitel	Muccini	Song	Kumar
Tool supported	✓ MESA	✓ Basic tool	X	✓ LTSA	X	X	X
Automatic algorithm	✓	X	✓	✓	✓	✓	X
Complexity of algorithm	Linear	various based on various factors	-	-	-	$O(V^3)$ V: set of events	-
Type of detected error	Process divergence, Non-local branching choice	Deadlock, Race condition, Timing condition	Conflicting behaviors	Implied scenarios	Non-local branching choice	Implied scenarios	Implied scenarios
Theory (T) or practical (P)	T/P	T mostly	T	T/P	T	T/P	T

2.4 Scenario based specification and message sequence charts

In software engineering, a practical approach is describing software requirements using scenarios. Scenarios are stories about people and interactions, including agents and actors and sequences of actions and event. In literature, there are reasons named for using scenario based designs for software systems [102]. Scenarios are descriptions that allow reasoning of use in situations even before actual implementations. They can describe different levels and details with multiple perspectives. For different purposes, scenarios can provide abstraction for the designer and problem solving. Also, they help managing consequences of changing designs [103].

The visual forms of scenario based specifications like UML Sequence Diagrams (SD) and Message Sequence Chart (MSC) can show the software behavior. Software system components (processes) or inter-object and inter-processes are shown using vertical lines with their interaction messages and send/receive events respectively in MSC or SD [104, 105]. These specifications can be used for model checking, formal verification, or monitoring for emergent behavior detection [106].

The MSC are visual and formal description techniques for software requirements and are widely used for Multi Agent Systems (MAS) and DSS [2, 3, 90, 107]. DSS that run on physically distributed computing devices are a class of software systems in which functionality and/or control are distributed. DSS has gained a broad attention in recent years for developing various systems and a main process in DSS is how to show the system components and their cooperation [2]. This process in many works is done utilizing MSC.

Development of MSC, its definitions and semantics has been standardized and published by Telecommunication Standardization Sector of International Telecommunication Union (ITU). MSC describes the communication behavior of system components and their environment in the

form of graphical representation and message interchanging. MSC grammar, semantics, and MSC textual presentation are defined in the ITU-T standards and can be related to Specification Description Language (SDL) [108].

Each MSC has a corresponding connectivity graph where nodes show the global system states (values of variables, state of execution of process and content of messages) [108]. MSC also can be used for real time systems with adding time constraints to events and specifying the time occurrence of each event. These constraints add more symbols like bold rectangles to the vertical lines of MSC. In later version of ITU-T Z.120 standard more constraints are added [109]. Other topics are related to showing MSCs to specify one system, instance decomposition, and rules of MSC documents in ITU-T Z.120.

MSC are extended to various forms for certain purposes [105]: Dynamic MSC [110], Triggered MSC [111], etc. A simple example of MSC is shown in Figure 1.

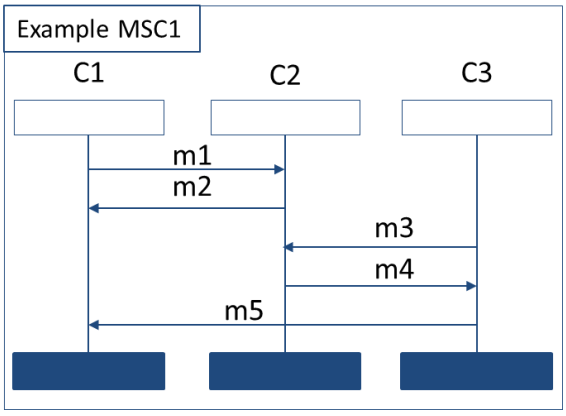

Figure 1: Example MSC showing interaction of three components

Each vertical line shows a component and the messages between them are shown with arrows. The name of the component is written above its line. Each message is shown with the label above it. The related scenario shows the MSC name.

A formal description of MSC from [112] is defined as a set (E, L, C, μ, <, M) where:

- E is the set of events consists of send and receive ones.
- L is a finite set of labels
- C is the set of components
- μ is the mapping of events to labels and components
- < is the set of total orders on the events and μ
- M maps the send events to receive ones

The graphical means for showing how a set of MSCs can be combined are shown either with Message Sequence Graph (MSG) or high level message sequence charts (hMSC). These two are structured models of MSC. MSG has operations such as choice and repetition. MSG has MSCs as its nodes and the edges represent the concatenation of MSCs. A choice or branching in MSG can be shown with a hexagonal. An hMSC is a graph like formation and each node can be either an MSC or an MSG. In hMSC, the start node is shown with an upside down triangle and the end MSCs is shown with a triangle. In many researches MSG and hMSC are considered almost the same. MSC, hMSC, and MSG can be considered as the early models of the system [14].

2.5 Social network analysis

Social networks (SN) as a basic view are a series of individuals and their interaction such as Facebook. However, many other forms of social networks exist. SNs can be kinds of interactions in different communities [113, 114]. The entities are considered as nodes with edges representing links between them. In different analysis the edges can be considered directed or undirected and with or without weights. Many analyses have been proposed to social networks to specify different characters and in various applications [115, 116]. The relation between nodes can be shown in matrices [117]

Some measurements in SN analysis are edge or node measures. This issue considers the edges of network (link between nodes). Some of these parameters are: Tie strength (defined for two nodes and depends on overlap of their neighbor and has specific extensions [118]), Betweenness (measures the extent that one node lies between the paths between other nodes), Degree (shows the direct links of a node), Closeness (measures closeness of one node to other nodes and shows how far it is from others), Authority (nodes that many other nodes point to them) and hub measure (the nodes that point to large number of authorities), exclusive measure (rare links belong to special nodes that other nodes do not have), etc. [119].

One application of SNA is verification of MAS communication in a network of agents. For example, some works evaluate and verify the communication patterns of MAS. They classify communications in classes like overloader, overloaded, isolated, and regular. Then they try to find design drawbacks in terms of communication patterns [120-122]. Data mining tools are developed for this purpose. SNA techniques like clustering is used for code debugging [123]. Some other data analysis tools are available that analyze the interactions among agents and visualize the result of running simulations to be compared with the modeling of agents in MAS [124].

There is much difference between these works with what is presented in this thesis. Many of the researches in this field are restricted to the verification of MAS from system logs, where the Multiagent system is implemented and deployed. Some like [124] work in the design phase, but they use some kind of simulation and run the models of the system to analyze the agents' interactions. The other factor that makes a difference between our work and other researches is that they use SNA for analyzing the communication to verify or detect different violation factors in terms of load balancing or business rules that must be satisfied. However, this work mostly looks for the violations in terms of new behaviors in the system (whether or not they are

accepted/declined by designers and stakeholder). However, these researches do not detect possible new behaviors (emergent behavior/implied scenario) and mostly investigate the communication performance, interaction verification, or issues associated with quality of service.

2.6 Model checking Multiagent systems (MAS model checking)

MAS model checking is a different approach for verifying multiagent systems (MAS) and programs via model checking. In this approach, the detection of emergent behaviors is not important. However, they check MAS against a specific property.

There are some approaches in this field [125] which mostly use logic or BDI (Belief, Desire, Intention) languages [126]. The former uses abstraction techniques to check a property about MAS. They mostly use CTL or TESL to express the system specifications. Moreover, the specification should be defined in logic format. Some works use ISPL as the input language of the system [127, 128]. Although the system is abstracted to solve the state space explosion problem, there are some other problems with these systems. One of the problems is specifying the formulae or property in logic which is prone to errors and is depending on expert intervention. If the specification is not defined properly, the property of the abstract system may hold while it may not hold for the original system. In addition, each specification requires a particular abstraction system, because it is content based. If the abstraction system is not defined properly, the results may vary or will not be valid. The other category applies model checking on logic programming languages with BDI architecture [126, 129, 130]. They use BDI logics and logic programming languages. They require translating these logics to other formats to use it as an input to model checking tools.

For both approaches, the agents' knowledge is specified. Moreover, the internal behavior of agents should be defined in the form of local states, variables, evolution functions, etc. Although each approach has its own advantages, they deal with the internal knowledge and states of each

agent to validate properties at knowledge level. However, when we verify MAS, we consider the agents as black boxes. In this research, we are looking for emergent behaviors and not validating agents' knowledge. Therefore, the category of MAS model checking is quite different with our approach. Our technique can be used for various AOSE methodologies and does not depend on a specific programming/logic language.

Another approach for checking the Multiagent systems is considered as debugging the codes of the developed system. In some methods, the design artefacts are used against the protocols of agents' interactions to verify the system such as Petri nets and semantic based approaches. In these works, the actual code of the MAS is required. In addition, the internal relations, beliefs, and knowledge of each agent must be identified, other than defining the interaction protocols [123]. The verification of MAS in this category can be done as intra-agent or inter-agent level. The intra-agent requires the agent model in terms of BDI, and the inter-agent level focuses on the interactions that occur during the execution of the MAS. The information used in the inter-agent level is in the form of messages in an Agent Communication Language (ACL). In this category the execution logs of the MAS are analyzed for the interaction verification [124]. The other similar field of research is verification of the MAS code using model checking approaches and is called program model checking, which uses the running system to construct the model [131]. A complete set of specification and verification of MAS can be found in [132].

2.7 Component based modeling

We present another category of works which is different from our work in this section. However, it deals with modeling the components in a component based system and in distributed systems and the ideas about the modeling and considering the interactions among components are valuable. We only present a brief introduction to one of the recent frameworks in this area called BIP and

leave the rest of the frameworks for component based to the reader. BIP provides a modeling language and a framework for the modeling of real time heterogeneous components. In this framework, the components are considered in a three layers' form: Behavior, Interaction, and their Priorities. The components consist of ports, control states, variables (to store data), and a set of transitions. The components' connectors and their interactions as well as the priorities between the interactions are modeled as well. Based on this modeling of the atomic components, the other components such as compound ones can be constructed. The framework also provides an execution platform to generate the code from this modeling using C++ TCP socket or MPI [133-136]. One of the main other works in this category is REO, that provides the channel based model for composition of components through connectors, instead of using glue code [137]. The main difference of this area of research (BIP) with our work is that it provides a model based on the states of the components and through this modeling and controlling the interactions among the components (using priorities) certifies the correctness of the implementation. On the other hand, we use the MSC and SD to certify the behavior of the software components and try not no model or use controller among the interactions and find out based on the visual interactions of components in the SDs whether an emergent behavior can occur.

2.8 Runtime verification

Runtime verification is referred to as analysis of the information of a running system during execution time and checking its behavior to verify it satisfying or violating some properties that are defined for the system. The properties are defined using logic languages. The execution of the system is logged and analyzed based on this logic to verify that the properties are satisfied. Different approaches and algorithms are developed for runtime verification [138, 139]. Since this is an area of work which is a lot different than our work, we only explain some high level concepts

and do not review the details of the works in the literature for runtime verification. One of the existing approaches for verification of software applications in the code level is model checking. In some cases, the model is extracted automatically form the code by statistically analyzing it [140].

Many of the runtime verification algorithms and approaches use the Message Passing Interface (MPI) for their verification or are specifically developed for verification of MPI programs. MPI is a communication protocol for parallel programming, which supports point to point and collective communications. MPI is a specification of the message passing interface and has various implementations as MPI libraries. The MPI helps to develop and run parallel programs according to its standard, when data is moved from address space of one process to another. It specifies cohesion and coupling strategies for multithreaded and thread safe interface. The MPI has eight basic functionalities as follows:

- Communicator: an object of this class connects processes in groups in an MPI session. It gives each of the processes an independent identifier and arranges them in an ordered topology.
- Point-to-point basics: It provides some functionalities that allow the communication between one process with another process. MPI_Send is one of these functions.
- Collective: The collective functions provides abilities for communication among all processes in a group. MPI_Bcast takes data from one process and sends it to the other processes in the group. The reverse functionality is provided by MPI_Reduce, which takes data from all processes in a group, performs an operation (e.g. summation), and returns the result to one node.

- Derived data types: This includes various data types for the communication of data between processes, such as MPI_INT, MPI_DOUBLE, etc.
- The other four concepts are restricted to MPI-2 only.
- One-sided communication: It provides three actions of write, read, and reduction operation to/from/on a remote memory.
- Collective extensions.
- Dynamic process management: Provides the ability for the MPI-processes to create new MPI-processes, or establish dynamic communication with MPI-processes that started separately.
- I/O: The parallel I/O or MPI-IO provides functionality to abstract the I/O management on distributed systems.

Model checking is used for the verification of MPI programs [141]. In many methods, specific functionalities of MPI, such as blocking or non-blocking or wait functions is investigated for the verification [142, 143]. One of the popular tools for the verification of MPI programs is ISP (in-situ partial order) [144]. ISP has a scheduler that controls the occurrence of functions of the MPI library based on the algorithm developed for verification of the program [145, 146]. It extracts the related interleaving of messages on processes and executes those interleaving messages. The ISP is one of the tool series that are developed for dynamic runtime verification of MPI programs at the University of Utah.

The main difference of runtime verification with our work is the phases that the behavior of the software should be certified. In the runtime verification, the implementation details is required, while in our work the designs in the form of MSCs or SDs are needed. We do not evaluate any of the works since they provide valuable properties in different spectrums of the software.

2.9 Summary

In this chapter the background study is provided. We have covered literature review on the software V&V, including formal methods and model checking approaches. Then different methodologies for the detection of emergent behaviors and the scenario based systems, specifically the Message Sequence Charts are studied. A category of works that are often confused with our work is the model checking of Multiagent systems. We have presented a brief summary of the works in this area of research and the differences with our research area. One main difference is considering the details of the internal knowledge space for the agents in this area, while we look at the agents as black boxes. The other two sections that we have studied are social network analysis and runtime verification. We use the concepts from social network analysis in our modeling. Also, we set up a Linux cluster in order to test our work on MPI programs, which was not feasible for our work, and contradicts with our assumptions. MPI is mostly used in runtime verification and a summary of the main functionalities of this library is included in this chapter.

Chapter Three: **Phase I: Modeling the behavior of the system components**

3.1 Introduction

In this chapter, we explain our approach to model the behavior of system components from the system designs. We have modeled the behavior of the system in a way other than the traditional approaches that use automata based modeling. In this modeling, we use the interaction graphs which is inspired by social networks. The reasons behind this modeling, our contributions, and the details of behavioral modeling with interaction graphs is explained in detail in this chapter.

3.2 Problem definition

One of the main research areas in the detection of Emergent Behaviors (EB) and Implied Scenarios (IS) in the Distributed Software Systems (DSS) is the behavioral modeling phase. There are many approaches on the synthesis of behavioral modeling of the system components that discuss how the scenarios should be modeled into state machines or transition systems in order to specify the behaviors defined in the scenarios. Although this modeling approach has its own advantages, there are some issues that makes the usage of this approach application specific or hard to use in all systems, especially in large scale DSS. A list of the problems with the researches that use behavioral modeling with state machines for the detection of EB/IS is listed previously. Some other are:

1. Only recently methodologies are devised to overcome the scalability of behavioral modeling by developing distributed algorithms that can handle the state space explosion problem, when the specified properties of the system or the requirements become large.
2. The system should be modeled again if the requirements are changed.
3. The states of each component are considered separately.

4. When the behavior of a component in the whole system is required, the states' model of the component from various scenarios should be integrated. This integration introduces another problem called "overgeneralization" [4, 42] that can present new behaviors – other than the ones specified in the system – in the model.
5. The behavioral model of the system is considered as executing the models of all components in parallel. This process requires specific semantics in order to prevent the existence of new behaviors or other problems that can be introduced in the system model.
6. The behavioral model of the individual components or the system does not contain the components' interactions directly. This case can be seen in the component behavioral modeling obviously, where the only part that relates to the messages sent/received by one component is considered as its transition between various states. However, even this case does not contain the information about its interaction to other components, and only considers the individual states on the single component.
7. The transition between the states of each component depends on the contents of the messages. This case is even used in the EB/IS detection process in later steps. Therefore, in many cases the modeling is either application specific or requires the domain expertise knowledge, which makes it hard to produce the whole process general or fully automated.

3.3 Methodology

To overcome some of the above mentioned problems, we tried to approach the behavioral modeling from a different perspective. Since one of the reasons of the EB/IS occurrence in DSS and Multiagent Systems (MAS) is the interactions among software components/agents, this concept is modeled in the system. To model the interactions among software components, the social networks and the approaches in modeling the interactions in a social network are considered.

Therefore, in our modeling, each scenario is modeled into a separate graph (network). The followings are a list of features that are considered in the modeling:

- Differentiating among various states of each component.
- Preserving the interaction information for each component.
- Preserving the interaction information among various components.
- Using the same model for both of the component and system level analysis. In other words, modeling the system once, and use it for analysis in both levels.
- Making the process of behavioral modeling fully automated and general other than application specific, without requiring information or domain knowledge from an expert.
- Try to avoid the overgeneralization problem.

In order to preserve the states of each component for the analysis phase, the individual states are modeled as the nodes of the graph. Also, we require a model that is general and includes enough information to cover both the component and system level analysis. To consider this feature, we model the graph vertices with two concepts: Core and Node. Each Node of the graph consists of a Core and some other features. Each Core, represents all the information which is required for component level analysis and contains the information of a component associated to the point of sending/receiving a message. This point in automaton theory can be interpreted as the events or the states of the components. We refer to this as a point of interaction or simply the states of the component. The information that is saved in each Core contains the message label (instead of message content) and the type of a message (send or receive), as well as the information about the name of the component and its next state (next Core). The Cores of the interaction graph are used for component level analysis. For system level analysis, the Nodes of the interaction will be used.

This information includes the interacting process name and interacting node, the information about its next states in each scenario, as well as the sender of the message that causes a transition in each of the states of each component.

It is worth mentioning that some of these features are added to detect specific types of EB/IS, and not all the features are used in the detection of all types of EB/IS.

The graph consists of a list of all Nodes. The edge (interaction) information is used as a data in each Node of the graph and is not modeled separately, since in each scenario, only one message is sent/received from each state and two interactions are not allowed from a single point.

This modeling is helpful in the detection of all classes of EB and IS that are determined in Phase III (Chapter Five). Moreover, the system level behavior can be considered in different scenarios of the systems and be analyzed by comparing the interaction graphs modeled from each scenario separately without requiring to make a complete model connecting all graphs.

Furthermore, the system level model is constructed from the beginning with the behavioral models of each component, without requiring any other action or completing other steps to have a behavioral model for the whole system. Conversely, the system level behavioral model in the current works in the literature is constructed after the behavioral models of each component is made. In these works, if the number of states for each component are large, we may face to the storage or computation problems when analyzing the behavior of the whole system (which is considered as the parallel execution of the individual behavioral models of all components). Also, in these works specific actions should be defined to connect the behavioral models of the components.

Another advantage of this modeling is the usage of existing structures for the graph storage data and computations.

In this approach, we use the information of each starting or end point of the message on each process line from the SDs or MSCs as the information that should be stored as Core or Node in the interaction graphs. We refer to these points as the states of each component in the system.

3.3.1 Basic definitions

3.3.1.1 Message Seuence Charts (MSC)

Definition 1. Communication set of one process $\sum_p$

Let $P = \{ p, q, r, \dots \}$ be the finite set of processes (components, agents) of the system that are interacting with each other using a finite set of messages M. We define $\sum_p$ as the set of communications that process p takes part in

$$\sum_p = \{p!\, q(m), p?\, q(m) | p, q \in P\, , m \in M\}$$

The $p!\, q(m)$ defines that process p sends message m to process q, and $p?\, q(m)$ defines that process p receives message m from process q.

Definition 2. Communication set of all processes $\sum = \cup_{p \in P} \sum_p$ (Alphabet)

We define $\sum = \cup_{p \in P} \sum_p$.

Definition 3. Message Sequence Chart

Each MSC $\mathcal{M}$ shows a visual form of processes P and their interacting messages over the finite set of messages M. An MSC $\mathcal{M}$ is a structure $\mathcal{M} = (E, P, M, \mu, \leq, \alpha)$ where:

$E = \{e_1, e_2, \dots\}$ is a finite set of $Send$ and $Receive$ events, where $S = \{s_1, s_2, \dots\}$ is the set of $Send$ events and $R = \{r_1, rs_2, \dots\}$ is the set of $Receive$ events.

P is a finite set of processes.

M is a finite set of messages.

$\mu: E \rightarrow \sum$ is a mapping function of events to messages and processes. For $p \in P$, $x \in \sum$ and $e \in E$:

$$E_{p\mathcal{M}} = \{e \mid \mu(e) \in \sum_p, \mu(e) = x\}$$

The set of events on process $p \in P$ in MSC $\mathcal{M}$ is:

$$E_{p\mathcal{M}} = \{e \mid \exists m \in M, e \in S, \mu(e) = p!\, q(m)\ or\ e \in R, \mu(e) = p?\, q(m)\}$$

And

$$E = \cup_{p \in P}\, E_{p\mathcal{M}}$$

$\leq$ is the set of total orders on E and μ.

$\alpha: S \rightarrow R$ maps the send events to receive events. For $e, e' \in E$ and $p, q \in P$, we define:

$$e <_{pq} e' \equiv\ \exists m \in M\ such\ that\ \mu(e) = p!\, q(m)\ and\ \mu(e') = q?\, p(m)$$

The $e <_{pq} e'$ relation explains that the message m sent by p at event e is received by q at event e'.

Definition 4. Local visual order $<_p$ (visual order of one process in an MSC)

Each MSC $\mathcal{M}$ has a visual structure showing the set of processes P interacting to each other via sending or receiving messages. The process $p \in P$ in an MSC is shown by a vertical line representing the life line of the process. The interacting messages are shown with arrows (edges C) from one process to another process. The events $E_{p\mathcal{M}}$ on process p have a ***local visual order*** represented by $<_p$, which is the total order of events on p as displayed in MSC $\mathcal{M}$.

Definition 5. Visual order (visual order of all processes in an MSC)

The ***visual order*** of an MSC $\mathcal{M}$ contains the local orders of all processes and the set of all edges C on that MSC:

$$(\cup_{p\in P} <_p) \cup C$$

3.3.1.2 High Level Message Sequence Charts (hMSC)

Definition 6. High level MSC $\mathcal{G}$

High level MSC (hMSC) is a structure $\mathcal{G} = (P, M, \boldsymbol{\mathcal{M}}, V, Ed, \mathsf{C}, F_0, F_f)$, where P is the set of processes, M is the set of messages, $\boldsymbol{\mathcal{M}}$ represents the set of MSCs, V represents the vertices, $Ed \subseteq V \times V$ is the set of edges, and C is a mapping function $\mathsf{C}: V \to \boldsymbol{\mathcal{M}}$. The $F_0 \subseteq V$ and $F_f \subseteq V$ are the initial and final vertices of $\mathcal{G}$.

Definition 7. Process's High level structure $\mathcal{G}_p$ (high level structure of one process)

For each process $p \in P$ in $\mathcal{G}$, we define a structure $\mathcal{G}_p = (M_p, \boldsymbol{\mathcal{M}}_p, V_p, Ed_p, \mathsf{C}_p, F_{0_p}, F_{f_p})$, where $\boldsymbol{\mathcal{M}}_p \subseteq \boldsymbol{\mathcal{M}}$ is the set of MSCs that p participates in, and has at least one action such that $\sum_p \neq \emptyset$. $M_p \subseteq M$ is the set of messages over $\boldsymbol{\mathcal{M}}_p$. V_p and $Ed_p \subseteq V_p \times V_p$ are the set of vertices and edges that are mapped by $\mathsf{C}_p: V_p \to \boldsymbol{\mathcal{M}}_p$. The $F_{0_p} \subseteq V_p$ and $F_{f_p} \subseteq V_p$ are the initial and final vertices over V_p as visually displayed in $\mathcal{G}$.

Definition 8. Global visual order $\beth_p$ (visual order of one process over its high level structure)

Each process $p \in P$ follows a visual order on its events $E'_p = \cup_{\mathcal{M}_p} E_{p\mathcal{M}}$ as displayed in $\mathcal{G}_p$. We define $\beth_p$ as ***global visual order*** of p over $\mathcal{G}_p$, which is the total order of events E'_p in $\mathcal{M}_p$.

3.3.1.3 Active processes/agents

Definition 9. Active process in an MSC $p_{a\mathcal{M}}$

Let processes $p, q \in P$ have some actions in an $MSC\ \mathcal{M} \subseteq \boldsymbol{\mathcal{M}}$. We define process p as an ***active process*** in $\mathcal{M}$ and refer to it as $p_{a\mathcal{M}}$, if for the first action of p in its local visual order $<_p$, the following condition is satisfied in $\mathcal{M}$:

$\exists m \in M, e \in S |\ \mu(e) = p!\, q(m)$.

Consider the MSCs in Figure 2. There are four MSCs in this example. The order of the execution of the MSCs is shown in the left side of the figure, in part (a). The hMSC is shown with a triangle as the starting point and a vertical triangle as the ending point. The nodes of the graph indicate the MSCs and their order of occurrence is shown with arrows between them. In this example, first MSC1 is executed. Then there is a branch to execute either MSC2 or MSC3, followed by MSC4 as the final MSC. The high level structure for a process comes from its functionality in each of the MSCs and the hMSC of the system. For example, the high level structure $\mathcal{G}_A$ (Definition 7) for component A, is the same as the hMSC of the system; since it has interactions in all MSCs of the system. However, the $\mathcal{G}_p$ for components B and C are different. The $\mathcal{G}_B$ is shown in part (b) of Figure 2 where it shows that component B is only involved in MSCs M1, M2, and M4. Part (c) of this figure represents that component C has interactions in MSCs M1 and M4.

Referring to Definition 9, component A is an active process in all MSCs except M2. In MSC2, component B is an active process.

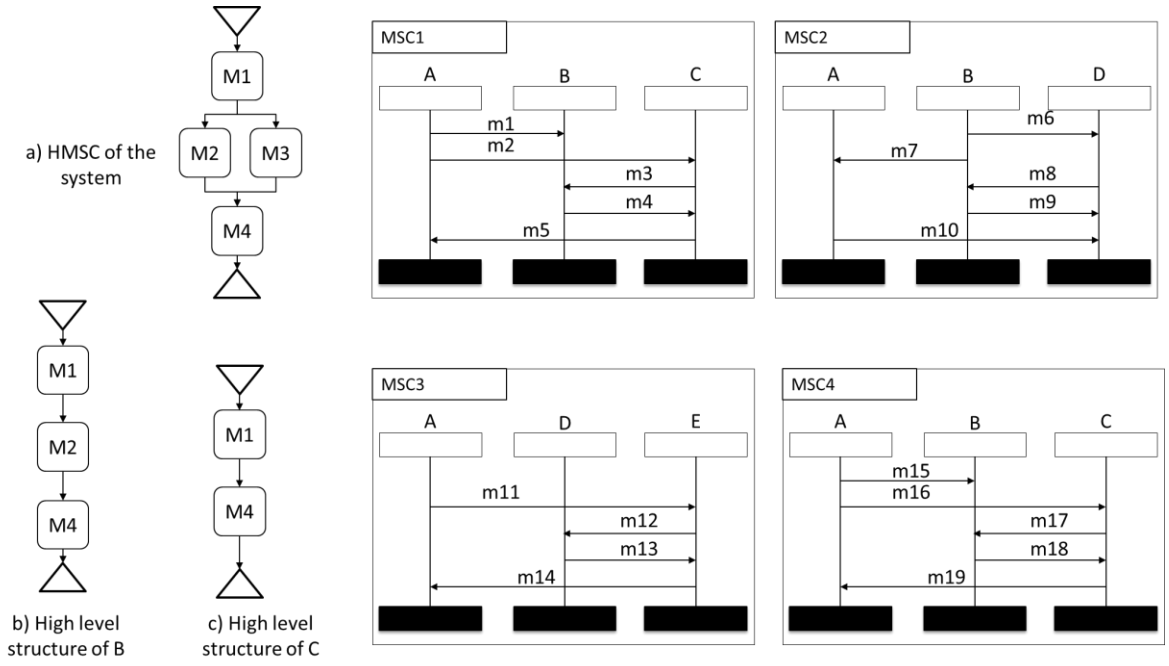

Figure 2. Four MSCs with the hMSC of the system in part (a) and high level structures of components B and C in (b) and (c)

3.3.2 Modeling definitions

3.3.2.1 Graphs

We will refer to the modeling that we defined earlier as the ***interaction graphs*** in this thesis. In order to specify the features of MSCs/SDs that should be modeled and used for analysis of EB/IS, the following definitions will be used.

Definition 10. Interaction graph G

An ***interaction graph*** is a directed graph defined as $G = (N, E, f)$ where N is the set of vertices that we refer to as Nodes, E is the set of edges between the vertices, and $f: E \rightarrow N$ is the mapping function that assigns each edge to its set of vertices. Each graph expresses the interactions between different processes in MSC $\mathcal{M}$ preserving its visual order.

Definition 11. Core C

Core $C = (Cont, Nbr, G, p)$ is the core information of each vertex V in the interaction graph G for process p and are derived from the information specified to the incoming or outgoing messages on the life line of component p in the related sequence diagram or MSC $\mathcal{M}$, without considering the information about the interactions of the specified state on component p. $Cont = \{Mlabel, MType\}$ represents the message labels and message types in each Core, where $MType = \{Send, Receive\}$ and $Mlabel$ is the labels of the messages, which is used instead of message contents. Nbr stores the data about who is the next state (Core). G identifies which graph (scenario) this Core belongs to.

Definition 12. Node N

Node $N = (Cont, Nbr, G, p, NCore, NInt)$ is the vertex of the interaction graph G for process p. Each Node contains the information about one state of component p and the interactions of the associated component in that specific state. The data derived for each vertex V in the interaction graph G is derived from the information specified to the incoming or outgoing messages on the life line of component p in the sequence diagram or MSC $\mathcal{M}$, considering the information about the interactions of the specified state on component p. $Cont = \{StateTransitionSender, IntP\}$ where $StateTransitionSender$ represents the State transitions of that state, i.e. the sender of the message that causes a transition in the current state, and $IntP$ represents the name of a process that Node N interacts with. The $NCore$ is the Core of the Node that contains the state information of that node. Nbr stores the data about who is the next Node (State). $NInt$ is the Node of the process that Node N interacts with.

To illustrate these definitions, consider the MSC in Figure 3. In this MSC, four agents are communicating through seven messages. Since the content of the messages are not important, the messages are labeled with m1 to m7.

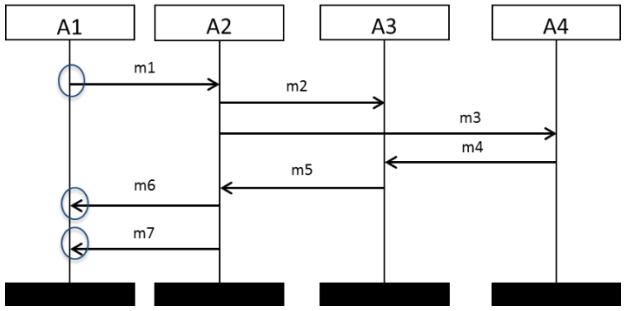

Figure 3. Example MSC

In our approach, we model the behavior of the agents from the scenarios and include all the information that can be extracted from the lifeline of the agents in sending or receiving a message. These points for agent A1 is shown with blue circles in Figure 3. This modeling requires information about the message labels that are sent/received in each point (state), its next state, which agent has send/received that particular message, in each state to which agent this agent (A1) interacts with, and who is the sender of the message that causes a change in the state of A1. Considering the interactions of A1 with A2, these are shown in Figure 4. In the right hand side, the communications of A1 and A2 and the message labels that are saved for each of them in their states is shown. In the left hand side, the states of A1 are demonstrated. We also include a link to show the next state of each state on A1.

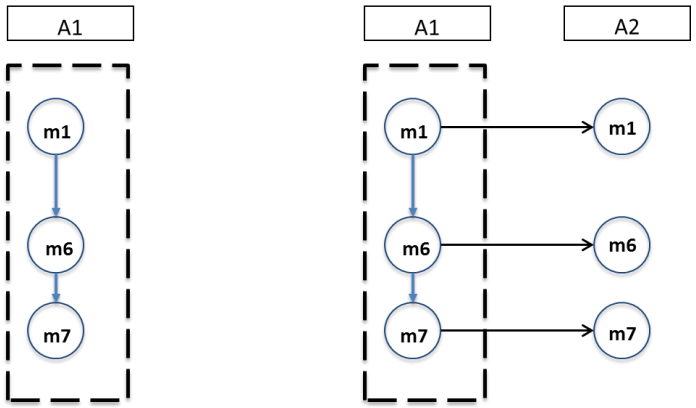

Figure 4. Part of interaction graph for agents A1 and A2

The complete interaction graph for MSC of Figure 3 is presented in Figure 5. The name of the agents above each vertical set of vertices represent part of the graph that is modeled as the behavior of each agent. In this figure, the Cores and Nodes are shown in separate colors. The graph is a list of Nodes in the implementation. As explained in the definitions, each Node contains Core as part of its definition. All the links shown in this figure (state transitions and interaction information) are shown as different pieces of information in the Cores and Nodes.

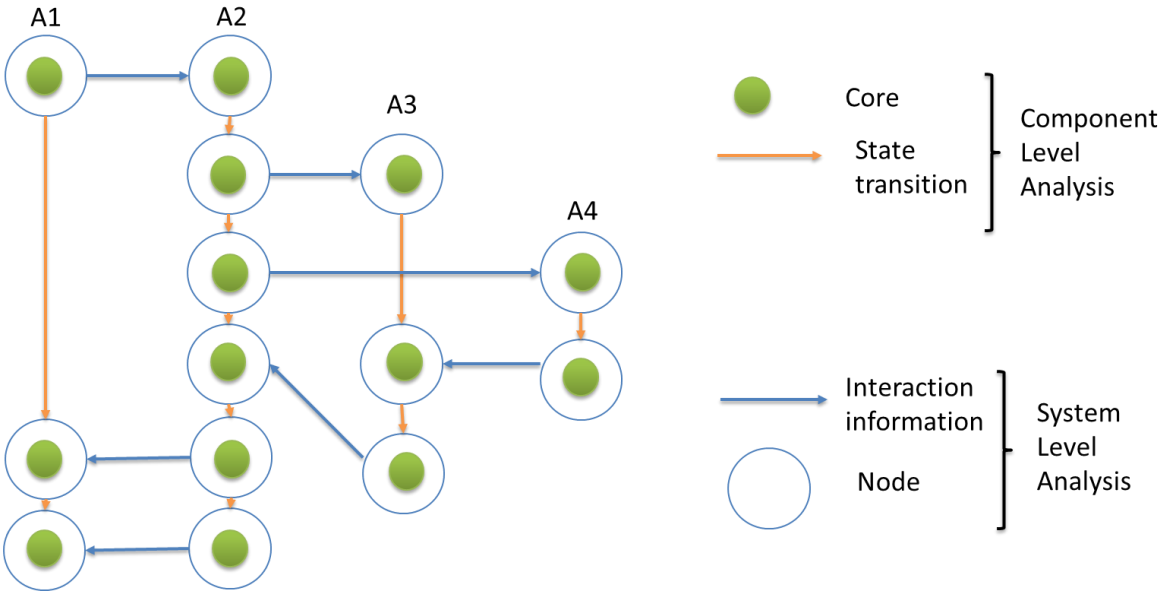

Figure 5. Modeled interaction graph for agents of Figure 3 containing Core and Nodes

The Cores, contain these set of information:

Content , Neighbors, Number, BelongToGraph, and ProcessName. The Content is {MessageContent (message label) + messageType (Send or receive)}.

And the information reserved in the Nodes are:

Content , Neighbors, Number, BelongToGraph, ProcessName, NodeCore, StateTransitionSender, InteractingProcessName, InteractineNode. In this set, the Content is {StateTransitionSender + InteractingProcessName}. The NodeCore represents the Core of a Node that contains the state information of that node. The StateTransitionSender represents the State transitions (the sender of the message that causes a transition in the current state), the InteractingProcessName is the name of a process that this Node interacts with, and InteractineNode represents the Node of another process that the current Node interacts with. All of these information can be found in Definitions 11 and 12.

3.3.2.2 Communications to detect EB/IS

Definition 13. Send vector of one process in an MSC $\delta_{p\mathcal{M}}$

We define a ***send vector*** $\delta_{p\mathcal{M}} = (a_1, a_2, \dots, a_n)$ for each process $p \in P$ in an MSC $\mathcal{M}$. The elements of $\delta_{p\mathcal{M}}$ can have one of the following values:

1. $if\ \exists q_i \in P, \exists s_j \in S, \exists e \in S\ and\ \exists m \in M\ such\ that\ \mu(e) = p!\, q_i(m) then\ a_i = s_j.$
2. $if\ \exists q_i \in P, for\ some\ s_j \in S, and\ for\ some\ e_k \in S\ and\ for\ some\ m_k \in$
   $M\ such\ that\ \mu(e_k) = p!\, q_i(m_k)\ and\ |\mu(e_k)| > 1\ then\ a_i\ is\ a\ set: a_i = \cup_j s_j.$
3. $if\ for\ q_i \in P, \nexists s \in S, \nexists e \in S, and\ \nexists m \in M\ such\ that\ \mu(e) = p!\, q_i(m) then\ a_i = \emptyset.$

The *send vector* δ_{pM} consists of n element where $n = |P|$. The $\delta_{p\mathcal{M}}$ is a set of sets $\{a_1, a_2, \dots, a_n\}$. The a_i is the set of all send events in which process p sends messages to another

process $q_i \in P$ in an MSC $\mathcal{M}$. The three cases above, represent that a_i can have exactly one member (case 1), more than one member (case 2), or be an empty set (case 3).

Definition 14. Set of all send vectors of one process in all MSCs it takes part δ_p

$\delta_p = \cup_{\mathcal{M}} \delta_{p\mathcal{M}}$.

Definition 15. Receive vector of one process in an MSC $\gamma_{p\mathcal{M}}$

We define a ***receive vector*** $\gamma_{p\mathcal{M}} = (b_1, b_2, \dots, b_n)$ for each process $p \in P$ in an MSC $\mathcal{M}$. The elements of $\gamma_{p\mathcal{M}}$ can have one of the following values:

1. $if\ \exists q_i \in P, \exists r_j \in R, \exists e \in R\ and\ \exists m \in M\ such\ that\ \mu(e) = p?\, q_i(m) then\ b_i = r_j.$
2. $if\ \exists q_i \in P, for\ some\ r_j \in R, and\ for\ some\ e_k \in R\ and\ for\ some\ m_k \in M\ such\ that\ \mu(e_k) = p?\, q_i(m_k)\ and\ |\mu(e_k)| > 1\ then\ b_i\ is\ a\ set: b_i = \cup_j r_j.$
3. $if\ for\ q_i \in P, \nexists r \in R, \nexists e \in R, and\ \nexists m \in M\ such\ that\ \mu(e) = p?\, q_i(m) then\ b_i = \emptyset.$

The *receive vector* $\gamma_{p\mathcal{M}}$ consists of n elements where $n = |P|$. The $\gamma_{p\mathcal{M}}$ is a set of sets $\{b_1, b_2, \dots, b_n\}$. The b_i is the set of all receive events in which process p receives messages from another process $q_i \in P$ in an MSC $\mathcal{M}$. The three cases above, represent that b_i can have exactly one member (case 1), more than one member (case 2), or be an empty set (case 3).

Definition 16. Set of all receive vectors of one process in all MSCs it takes part γ_p

$\gamma_p = \cup_{\mathcal{M}} \gamma_{p\mathcal{M}}$.

Definition 17. State vector of one process in an MSC $\beta_{p\mathcal{M}}$

We define a ***state vector*** $\beta_{p\mathcal{M}} = (\cup (\delta_{p\mathcal{M}}, \gamma_{p\mathcal{M}}), <_p) = (\tau_1, \tau_2, \dots, \tau_z)$ for each process $p \in P$ of an MSC $\mathcal{M}$ where $z = |E_{p\mathcal{M}}|$. The state vector $\beta_{p\mathcal{M}}$ is the total order set of $E_{p\mathcal{M}}$ in the MSC $\mathcal{M}$ with respect to its local visual order $<_p$:

$$\tau_i = e_i \in E_{p\mathcal{M}}, \exists m \in M, such\ that\ e \in S, \mu(e) = p!\, q(m)$$

$$or\ e \in R, \mu(e) = p?\, q(m)$$

Definition 18. Set of state vectors of one process in its high level structure β_p

For process p we define $\beta_p = \cup_{\mathcal{M}}\ \beta_{p\mathcal{M}}$ with respect to its global visual order $\beth_p$ over $\mathcal{G}_p$.

Definition 19. State transition vector of one process in an MSC $\varphi_{p\mathcal{M}}$

The ***state transition vector*** $\varphi_{p\mathcal{M}} = (\sigma_1, \sigma_2, \dots, \sigma_x)$ for each process $p \in P$ of an MSC $\mathcal{M}$ where $x = |E_{p\mathcal{M}}|$, represents the senders of messages that cause a transition in states of process p in the MSC $\mathcal{M}$. Each action on the life time of process p in MSC $\mathcal{M}$ is either $p!\, q(m)$ or $p?\, q(m)$ for $m \in M$. For $m \in M$ and $\forall e \in E_{p\mathcal{M}}$:

$$\sigma_i = p\ if\ \mu(e) = p!\, q(m)\ or\ \sigma_i = q\ if\ \mu(e) = p?\, q(m)$$

Definition 20. Set of state transition vectors of one process in it is high level structure φ_p

We define $\varphi_p = \cup_{\mathcal{M}}\ \varphi_{p\mathcal{M}}$ as the total set of state transitions for process $p \in P$ over the set of $MSCs\ \boldsymbol{\mathcal{M}}$ in $\mathcal{G}_p$ with respect to its global visual order $\beth_p$.

Definition 21. Interaction vector of one process in an MSC $\omega_{p\mathcal{M}}$

We define an ***interaction vector*** $\omega_{p\mathcal{M}} = (w_1, \dots, w_z)$ over the state vector $\beta_{p\mathcal{M}} = (\tau_1, \tau_2, \dots, \tau_z)$ that represents the interaction details for process p in MSC $\mathcal{M}$ where $z = |\beta_{p\mathcal{M}}|$ and $w_i = (s_i, r_i)$ and

1. $if\ \tau_i = e_i \in E_p\ such\ that\ \mu(e) = p!\, q(m)\ then\ s_i = p\ and\ r_i = q.$
2. $if\ \tau_i = e_i \in E_p\ such\ that\ \mu(e) = p?\, q(m)\ then\ s_i = q\ and\ r_i = p.$

Note that since the events E_p on process p have a *local visual order* $<_p$, the $<_p$ relation is preserved in the send, receive, and state vectors $\delta_{p\mathcal{M}}$, $\gamma_{p\mathcal{M}}$, and $\beta_{p\mathcal{M}}$ in the MSC $\mathcal{M}$. For the same reason, δ_p, γ_p, and β_p follow $\beth_p$.

Definition 22. Shared states of one process over two or more of its state vectors Θ_p

We define the ***shared (similar) states*** of process p over state vectors $\beta_{p\mathcal{M}_1} = (\tau_1, \tau_2, \dots, \tau_z)$, …, and $\beta_{p\mathcal{M}_\mathrm{n}} = (\tau'_1, \tau'_2, \dots, \tau'_y)$ of MSCs $\mathcal{M}_1$, …, and $\mathcal{M}_\mathrm{n}$ in $\mathcal{G}_p$ as $\Theta_p = (\theta_1, \dots, \theta_n)$ such that Θ_p is the set that $\Theta_p \subseteq \beta_{p\mathcal{M}_\mathrm{i}}$ where $i \in \{1, \dots, n\}$. Then we have the following: $\Theta_p = (\theta_1, \dots, \theta_n) = (\tau_w, \dots, \tau_x) = \cdots = (\tau'_a, \dots, \tau'_b)$.

Considering two state vectors $\beta_{p\mathcal{M}_1}$ and $\beta_{p\mathcal{M}_2}$, there can be multiple sets of states that are shared among these two vectors.

Definition 23. Set of all shared states of one process Θ'_p

Process p can have shared states of various lengths in the set of MSCs in $\mathcal{G}_p$. Therefore, we define $\Theta'_p = \cup\, \Theta_p$ as the ***set of all shared states*** of process p.

Definition 24. Shared state transition vectors of one process $\varphi'_{p\mathcal{M}1}$ (state transition vectors for the shared state vectors)

Consider two state vectors $\beta_{p\mathcal{M}_1} = (\tau_1, \tau_2, \dots, \tau_z)$ and $\beta_{p\mathcal{M}_2} = (\tau'_1, \tau'_2, \dots, \tau'_y)$ of MSCs $\mathcal{M}_1$ and $\mathcal{M}_2$ in $\mathcal{G}_p$ for process p and $\Theta_p = (\theta_1, \dots, \theta_n) = (\tau_w, \dots, \tau_x) = (\tau'_a, \dots, \tau'_b)$ as their shared state vector. We define ***shared state transition*** vectors $\varphi'_{p\mathcal{M}1}$ and $\varphi'_{p\mathcal{M}2}$ as the state transitions of $(\tau_w, \dots, \tau_x)$ and $(\tau'_a, \dots, \tau'_b)$ respectively.

Definition 25. Shared interaction vectors of one process $\omega'_{p\mathcal{M}1}$ (interaction vectors for the shared state vector)

We define ***shared interaction*** vectors $\omega'_{p\mathcal{M}1}$ and $\omega'_{p\mathcal{M}2}$ as the interaction vectors of $(\tau_w, \dots, \tau_x)$ and $(\tau'_a, \dots, \tau'_b)$, respectively.

In Chapter Seven various examples for the definitions of this section can be found.

3.4 Summary

In this chapter, we provided the required definitions for modeling the behavior of the components of the system from MSCs/SDs. We model the specifications of the components in the scenarios using interaction graphs. For each vertex of the graph, we consider two main concepts: Core and Node. The Core preserves the information about the component in the modeling, and the Node saves extra information related to the interactions of the components in each Core. The main purpose behind this modeling is being able to model the system once, by reading the scenarios from the input and extract the required information for component or system level analysis as needed. The Core is used for component level analysis and the Node is used for system level analysis. Other definition such as state transition vectors are also defined. These definitions will be used later for the detection methodology.

Chapter Four: **Phase II: Pre-Processing: remove components with no EB**

4.1 Introduction

In this chapter, we explain our devised methodology to reduce the scale of the problem that is under analysis. This phase is considered as a pre-processing phase of the component level analysis. One of the advantages of this method is its usage in the current approaches. Therefore, one can proceed by other methodologies to analyze a system for EB/IS detection by first applying this pre-processing to reduce the number of components that need to be verified. Therefore, the scale of behavioral modeling will be reduced and it results in the reduction of computational analysis. In the following the details of this approach will be explained.

4.2 Problem definition

One of the main problems in the behavioral modeling is known as the state space explosion problem. In synthesis of behavioral modeling, when the number of states of the individual components or the whole system increases, the number of states that should be considered in parallel to demonstrate the concurrent behavior of the system grow exponentially. The state space explosion problem restricts the number of states under analysis, and therefore restricts the size of the system that should be verified. For this reason, many approaches in the modeling the behavior of the system are developed including bounded model checking. In this approach, a finite prefix in the paths are analyzed and the length of the prefix is bounded by a number. Unlike the previous methodologies, not all the paths are investigated. Therefore, if the prefix is not defined properly, there is a chance to lose part of the analysis; since it cannot represent the system properly.

The other approaches that are developed to deal with the state explosion problem are symbolic model checking, partial order reduction, abstraction techniques, and counter example guided abstraction refinement. Each of these techniques require specific conditions to be applied.

The partial order reduction technique the exchange techniques are developed to substitute the enabled transitions of the system with its representative subset. In other words, the technique tries to reduce the number of independent interleavings of the concurrent processes. Also, in the abstraction technique, the system should be simplified and the simplified properties are not representing the exact system properties. Therefore, they may need iterative refinements. The refinement process is also required for the counter example guided abstraction refinement technique.

Based on the above mentioned explanations and the specific refinements for each of these techniques or the fact that they will not actually analyze all the possible paths of execution in the system, we look at the problem by a novel approach to detect components that have no emergent behavior in the system. Although this method will not reduce the scale of the problem in many works with a significant rate, it has shown up to 40% of reduction in the number of components that require to be modeled in other systems.

4.3 Methodology

To approach the state explosion problem, we consider the components that require the analysis in the whole system. In our technique, the components of the system that have no emergent behavior will be detected first and removed from the component level analysis. Therefore, the number of components and consequently the number of states under investigation reduces. This phase is considered as a pre-processing phase, before the behavioral model of the system is fully constructed.

The components that will not show an emergent behavior in the system are the components that behave or act with the same functionality in all scenarios of the system. These are the usually the components that have the exact functionality in all scenarios, always receive a message in the

system without having a send interaction, or the combination of their behavior are the same in most of the scenarios with minor differences in the other ones.

We have devised algorithms to detect these components both in synchronous and asynchronous communications to manage the scalability of the system.

4.3.1 Detect components with no EB in synchronous communication

For the detection of the components with no EB, we should consider their behavior in all of the scenarios of the system they are involved in. For this purpose, we consider vectors that represent different information for the functionality and interaction of each component separately. This information should be extracted from the scenarios in order to check the following conditions related to the component's interactions as described below:

1. The component has the exact functionality in all scenarios.
2. The component always receives a message in the system without having a send interaction.
3. The functionality of the component is the same in most of the scenarios with minor differences in at most one of the scenarios. The process in this case is an active in at most one scenario and is a passive process in all other scenarios it is involved in.

Either one of these conditions are satisfied, the component can be declared as a safe component with no EB at the component level.

Based on the catalogue of emergent behaviors the main reason for a process to show a new behavior is the conditions in which it is sending a message to the other processes. Therefore, if the process is not sending any message, it is always acting as a passive process and cannot emerge a new behavior by itself. Also, if the process is only sending a message in one of the scenarios of the system, it means that the process does not have any shared states in which it acts as an active process. Therefore, it cannot show an emergent behavior in the component level. These are the

rationales of checking conditions two and three. The first condition specifies the exact behavior of the component all over the system. The exact behavior of a component is specified not only as the messages it sends or receives (the states on its life line), but also as the interactions of the component with the other components. Therefore, for example if the component $C1$ is sending message $S1$ to component $C2$ in all of the scenarios, then it is considered as having the same behavior for component $C1$.

For checking these conditions, the following definitions are required.

Definition 26. Similarity of two vectors

$Sim(v_{ik}, v_{ij})$ is defined as the similarity of two vectors v_{ik} and v_{ij}. Vector v_{ik} shows the vector associated to process i in MSC k. Two non-zero vectors are similar *iff* their size is the same and their ordered elements are the same. In other words:

$$if\ v_{ik} = (e_{1k}, \dots, e_{mk})\ and\ v_{ij} = (e_{1j}, \dots, e_{zj})$$

then m=z and $e_{nk} = e_{nj}\ \forall\ n \in \{1, \dots, z\}$

The devised algorithm for the detection of components with no EB is as follows in Algorithm 4.I.

Algorithm 4.I: Detecting component with no EB in component level

Input: Graphs

Output: List of components that can be omitted for behavioral modeling (components with no EB at component level)

1. For each graph G
2. For each process p
3. Extract the send vectors $\delta_{p\mathcal{M}_i}$ from all graphs
4. If $\delta_{p\mathcal{M}_i} = \emptyset$ in all graphs
5. Report process p as No EB Process
6. Break
7. End if

8. If $\delta_{p\mathcal{M}_i} \neq \emptyset$ in at most one graph
9. Report process p as No EB Process
10. Break
11. End if
12. Extract the message labels for process p
13. Check the message labels for process p in all other graphs
14. If the message labels are the same in graphs
15. Add component p to similar components set $S_{imilarM}$
16. End if
17. End for
18. End for
19. For each component p in set $S_{imilarM}$
20. Extract the send vectors $\delta_{p\mathcal{M}_i}$ from the related graphs
21. Consider the send vector $\delta_{p\mathcal{M}_j} : \forall \mathcal{M}_j \epsilon \mathcal{M}$
22. Compute $Sim(\delta_{p\mathcal{M}_i}, \delta_{p\mathcal{M}_j}) \forall \mathcal{M}_j \epsilon \mathcal{M}$ in $S_{imilarM}$
23. If $\delta_{p\mathcal{M}_i}$is similar to $\delta_{p\mathcal{M}_j} : \forall \mathcal{M}_j \epsilon \mathcal{M}$ (1≤ j≤ n)
24. Add component p to similar components set $S_{imilarS}$
25. End if
26. End for
27. End for
28. For each component p in set $S_{imilarS}$
29. Extract the receive vector $\gamma_{p\mathcal{M}_i}$ from the related graphs
30. Consider the receive vector $\gamma_{p\mathcal{M}_j} : \forall \mathcal{M}_j \epsilon \mathcal{M}$
31. Compute $Sim(\gamma_{p\mathcal{M}_i}, \gamma_{p\mathcal{M}_j}) \forall \mathcal{M}_j \epsilon \mathcal{M}$ in $S_{imilarS}$
32. If $\gamma_{p\mathcal{M}_i}$ is similar to $\gamma_{p\mathcal{M}_j} : \forall \mathcal{M}_j \epsilon \mathcal{M}$ *(1≤ j≤ n)*
33. Add component p to similar components set S_{imilar}
34. Report as No EB process
35. End if
36. End for

Starting from the first graph, we extract the send vector of each process in all the scenarios the process is involved in. If the send vector of the process is empty in all graphs (line 4) or the send vector of the process in non-empty in at most one graph (line 8), then the process is reported as a component that cannot emerge a new behavior in component level. These are satisfying conditions two and three. Based on the catalogue of emergent behaviors the main reason for a process to show a new behavior is the conditions in which it is sending a message to the other

processes. Therefore, if the process is not sending any message, it is always acting as a passive process and cannot emerge a new behavior by itself. Also, if the process is only sending a message in one of the scenarios of the system, it means that the process does not have any shared states in which it acts as an active process. Therefore, it cannot show an emergent behavior in the component level. These two conditions are checked at first to eliminate the components with the specified behavior before checking and extracting the other vectors for these components.

If none of these conditions are satisfied, the message labels for the process are extracted in all graphs (scenarios). For each process, if the message labels are the same in all of the graphs, then the process is added to the set $S_{imilarM}$ (line 15). The message labels show the functionalities and the messages that a process sends and receives in each scenario of the system. The message labels can be interpreted as states of the process. If the message labels are the same in all scenarios that a component is involved in, then it is possible that the process behaves the same in all scenarios of the system (condition one). However, since the interactions are important in the behavior of a component in the system, we cannot be sure about the behavior of the component in terms of not having an emergent behavior. Therefore, the send and receive vectors of the component are checked against the other send and receive vectors in all other scenarios (lines). If the send vectors are similar to each other, and the receive vectors in all scenarios are similar to each other, then the component has the same behavior and functionality in all of the scenarios of the systems and it is reported as a component with no EB (line 34), since the send and receive vectors can represent what is sent to which process or received from each process respectively. The sender or receiver process has the same index as the index of the element in these vectors. An earlier version of this algorithm is published in [147].

4.3.2 Detect components with no EB in asynchronous communication

4.3.2.1 Asynchronous communication style

We have previously published this part in [148]. When we model the system in asynchronous communication in MSC, the components are considered as independent ones. It means that there is no control over the receive messages. Therefore, there is no wait function for the components when they accomplish their tasks in an MSC. Therefore, components may proceed to the next MSCs while the other ones are still executing the previous MSC [95]. This is implemented as wait functions or blocking send/receive functions in synchronous communication. In an MSC, asynchronousness means that last events of one component in an MSC is concatenated to its first event in the next MSC (in its hMSC) [37].

Also, there is no control on the receive events of a component and only senders are considered for the events [37]. Therefore, causal orders such as $r \rightarrow r$ and $s \rightarrow r$ are not valid (s and r represent send and receive events on **one** component). In this communication style, only $r \rightarrow s$ orders are valid. Therefore, we only are sure that the send message is sent after a receive message. But, the order of receiving two or more messages or sending a message with respect to a received message are not guaranteed. An example of these order are shown in the right side of Figure 6. The bold arrows show the asynchronous orders and other ones are the causal orders in synchronous style. The causal orders of asynchronous style are preserved in the synchronous style.

4.3.2.2 Properties for detecting components with no emergent behavior

Three conditions are explained in [37] where the implementation model is different from the scenario specification. First, one component has some shared states in receiving same events. Second, when the component has no control over its receiving messages. Third, the existence of non-local branching choices explained in [17]. The last condition is categorized in the catalogue

of emergent behaviors and more explanation can be found in Chapter Five. These differences may or may not cause an EB/IS and they should be analyzed based on the conditions specified in the EB Catalogue. Here, we only consider the first two conditions since we are investigating the issue in component level. Explanation can be found in next.

4.3.2.2.1 Checking the first property: shared states

The shared states have various definitions in the literature such as identical states [4]. We also provided the definitions for the shared states in the previous chapter. If the component has no shared states, it will not have an emergent behavior. However, we should check whether the component has no shared states or if it has, the senders are also the same and the message labels that the component sends after its shared states are the same. The details of these situations are explained in Chapter Five.

4.3.2.2.2 Checking the second property: order of received messages

In this part, the local visual order $<_p$ and semantic causality of the events should be analyzed. Based on the timing order of the messages and its meaning in an MSC (higher message occurs sooner), a causal-order graph for each MSC can be defined to show the order of the appearance of all the events [37]. However, this order is not reserved in the implementation in asynchronous communication style. The reason is the lack of control on the receive events. In Figure 6, the interactions between two components is shown. There are four messages which are shown by the send and receive events s and r on each of the components $C1$ and $C2$. The causal order graph for this figure (based on definitions in [37]) is shown in the right side of Figure 6. The order of the implementation is shown with bold arrows.

One other concept is semantic causality [4] which is associated to message contents and their semantics. For example, *"inserting a card"* is the semantic cause of *"entering pin"* from an ATM

machine which is shown by *"insert card" →" entering pin"*. In other words it means the occurrence of the second event depends on the first event. Semantic causalities should be defined by the domain expert and depend on the application.

Consider the events of component $C1$ in Figure 6**:** two send events s_2 and s_4 and two receive events r_1 and r_3. If semantic causality like $s_2 \rightarrow s_4$ is defined, we will not have EB when the order of receive messages change. It means the component has control on sending its messages. If we do not have semantic causalities, EB can occur. For example, if r_3 is received sooner than r_1, $C1$ can send s_4 before s_2. The other case is lack of a semantic causality between send events (but orders like $r_1 \rightarrow s_2$ are reserved). Here, the sending order is forced after a message is received. However, there is no control over the receive messages. Therefore, if the situations are satisfied, an EB can occur because the send orders may change.

Detecting the existence of an EB can be accomplished by analyzing the semantic and order causality for the events of each component. These parts are previously published in [148].

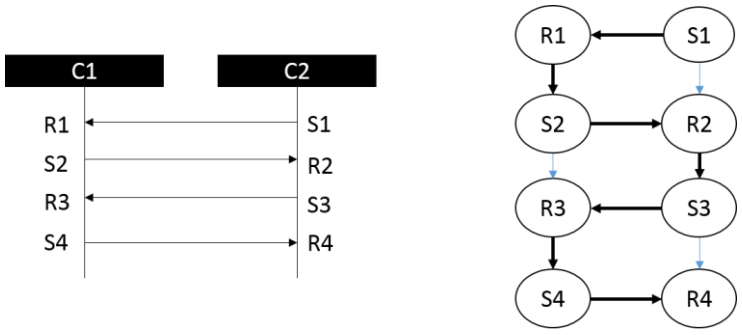

Figure 6. Events of a component and its corresponding causal order graph [148]

We define the causal order for each component as the following.

Definition 27. Causal order

The causal order of process p in MSC $\mathcal{M}$ is shown by a vector $CausalO_{p\mathcal{M}} = (caO_1, \dots, caO_n)$ which is the state vector of the process $\beta_{p\mathcal{M}}$ (Definition 17 chapter 3) when the s or r is added to each entry.

4.3.2.3 Algorithms

Two algorithms are devised to check each of the two properties explained above. The first algorithm checks the shared states and the second algorithm applies the second technique for each component. If the component is printed as the output of Algorithm 4.III, then it is a safe component and may not emerge a new behavior in asynchronous communication style in the component level.

Algorithm 4.II: Checking components for shared states

Input: state vector $\beta_{p\mathcal{M}}$, state transition vector $\varphi_{p\mathcal{M}}$, send vector $\delta_{p\mathcal{M}}$

Output: Components with no emergent behavior in first category

1. for each process p
2. $sharedStates_{p\mathcal{M}} \leftarrow FindSharedStates(\beta_{p\mathcal{M}})$
3. if ($NextState(\max(sharedStates_{p\mathcal{M}})) \in \delta_{p\mathcal{M}}$
4. if ($DifferentSenders(sharedStates_{p\mathcal{M}})$)
5. Remove(p)
6. end if
7. if ($DifferentLabels(NextState(max(sharedStates_{p\mathcal{M}})))$)
8. Remove(p)
9. end if
10. else
11. add p to $data$
12. end if
13. end for
14. Print ($data$)

The output of Algorithm 4.II is used as the input of Algorithm 4.III to check the causalities for the components.

Algorithm 4.III: Checking components for causality

Input: $data$ from Algorithm 1, semantic causalities $CausalS_{p\mathcal{M}}$, causal order $CausalO_{p\mathcal{M}}$, Send and receive vectors $\delta_{p\mathcal{M}}$ and $\gamma_{p\mathcal{M}}$

Output: Components with no emergent behavior in second category

1. for all components $p \in data$
2. if ($CausalS_{p\mathcal{M}} = \emptyset$)
3. Remove (p)
4. end if
5. if ($CausalO_{p\mathcal{M}} \in CausalS_{p\mathcal{M}}$) then
6. Print (p)
7. end if
8. if ($\delta_{p\mathcal{M}} \in CausalS_{p\mathcal{M}}$) then
9. Print (p)
10. else if (($\delta_{p\mathcal{M}} \notin CausalS_{p\mathcal{M}}$) && (($\gamma_{p\mathcal{M}} \rightarrow \delta_{p\mathcal{M}}$) $\in CausalS_{p\mathcal{M}}$))
11. Remove(p)
12. end if
13. end for
14. Print ($data$)

In the first step of Algorithm 4.II, we find all shared parts among all of the state vectors of each component. As explained previously, one component can have multiple sets of shared states in various MSCs it participates in (line 2). The next step is to check the next state of the last shared states for these sets (line 3). The component can emerge a new behavior if in these states (next state after its shared states) it is a sender and there are different senders for its shared states. Therefore, it should be removed (lines 4-5). If the senders are not different in these states but the message labels it sends in these states are not the same, it shows a branching choice (explained more in next chapter) and can have an EB and should be removed (line 8). The other components are sent for further analysis in the second algorithm.

The causalities are checked in Algorithm 4.III. In case of lack of a semantic cause or lack of a semantic cause for send events an EB can occur. These components should be removed (line 2 and line 10-11). Else, if it has semantic causalities defined between its send messages or causal orders of the component are part of its semantic causalities set, it will not have an EB and is printed as a safe component (lines 5-9). The output of the algorithm is the list of components with no EB

in asynchronous communication. We have published another version of the algorithms as well as the methodology in [148].

4.4 Summary

We use a pre-processing phase in order to find the components that will not show an emergent behavior in the component level. This will be a helpful stage in other researches as well, since it reduces the number of components that require behavioral modeling. In this chapter, we have explained the algorithms for the detection of such components in synchronous and asynchronous communication styles. One application of this approach is detecting neutral agents in a network of heterogeneous software components that we have published in [149].

Chapter Five: **Catalogue of emergent behaviors and implied scenarios**

5.1 Introduction

In this chapter, we explain the catalogue of emergent behaviors and implied scenarios. We have studied the common problems in DSS and MAS from the literature to find the common EB and IS that exist in these systems. In this thesis, we refer to the component level unexpected behaviors as *Emergent Behaviors* (EB) and *Implied Scenarios* (IS) are used for system level unexpected behaviors. The occurrence of EB and IS has different reasons and various conditions must be satisfied for the occurrence of an EB or IS in the system. Based on these criteria, we have classified each of the EB and IS into four sub-classes which are explained in detail in this chapter. Through the classification of EB/IS, we have introduced a new type of EB that can happen in MAS. This type is ignored in other cases, since the interactions among the agents are not considered. The detection of this type of EB cannot be done with the existing methodologies.

The catalogue of emergent behaviors and implied scenarios can help to have a general framework for the comparison of different detection approaches. This comparison framework contains the number of common types that can be detected in each category, and the process of behavioral modeling required for the detection of component and system level unexpected behaviors, as well as the computation and the scalability of the algorithms developed for each methodology. In the literature, there is no comparison in terms of the types of EB/IS that each methodology can detect. Most of the comparisons are on the computational costs. Even there are critics on the works that claim to detect all of the EB/IS that exist in a system [41].

Another advantage of the catalogue of EB/IS is providing solution repositories for each detected problem which is a flaw of the existing works [37]. General (Not application specific) solutions can be provided only when the origins of the problem are investigated. Since in this

catalogue various reasons and conditions of the occurrence of EB/IS are studied, developing solutions and general guidelines is possible. These conditions are explained for each class of emergent behavior and implied scenario separately.

The rest of the chapter is dedicated to the common problems in component level and system level. In each class, we have explained the specification of each separate sub-classes followed by the reasons of happening of each of them.

5.2 Component level emergent behaviors

The component level emergent behavior (CLEB) class is divided into four sub-classes in order to specify the conditions that must be satisfied for the occurrence of each type. The four classes are:

1. CLEB-I: Shared states
2. CLEB-II: Respond to different components
3. CLEB-III: Local branching choice
4. CLEB-IV: Race conditions

In the first class, some states of one component are identical (from the point of view of the component) in different scenarios of the system. These states can be referred as shared states, since they are some states which exist in multiple scenarios. The introduction of the second class is one of the contributions of our work [150]. In this class, the component can emerge a new behavior when it has shared messages coming from various agents, however, the process has the option to send the response to each of the senders. In this class, the interaction of the software agents is important. The third class specifies when the component has the option to choose between various branches. In other words, each of the next states become optional for the component. In the last class, we specify a case that is referred to as race conditions in this catalogue. The specification and details of each class are presented in this chapter.

5.2.1 CLEB-I: Shared states

In synthesizing the behavioral model of a component [4, 151], the states of one component might be shared in various scenarios. These states are merged as one series of states starting from the initial state of the component, when merging the behavior of one component from different scenarios. We refer to these states as shared states of one component in this work. The next states after these shared states in each scenario create branches in the behavioral model of the component. These branches make the behavior of one component nondeterministic. Therefore, in model checking, there are different techniques to make this behavior deterministic and then verify whether a component can show an emergent behavior [13]. However, not all of the branches and shared states for one component can lead to an emergent behavior in the system. The specifications and conditions that can lead to an emergent behavior in the existence of shared states are defined in this section. In general, sending a message from the component that has shared states can lead to a potential emergent behavior; since the process have the freedom of sending a message, if no restriction condition is applied on its sending messages. Therefore, if sending of the messages in these shared states is not controlled, the component may emerge a new behavior or forces a message to other components that can cause an implied scenario.

We refer to these types of emergent behaviors as **CLEB-I** and name it as shared states in component level, because process p may show an unexpected behavior when it has some shared states in two or more of the MSCs that it is involved in. In this section, the details of this type of emergent behavior caused by shared states of one process are explained in detail.

NOTE: In these cases, we consider the synchronous communications. The asynchronous communication or the case that the receive message is not received in the specified order are not

considered here. The change in the order of the receiving messages are studied in next sections (system level).

5.2.1.1 Specification

We have defined the shared states of one process in Definition 22, section 3.3.2. This definition is repeated here.

Definition 22. Shared states of one process over two or more of its state vectors Θ_p

We define the ***shared (similar) states*** of process p over state vectors $\beta_{p\mathcal{M}_1} = (\tau_1, \tau_2, \dots, \tau_z)$, …, and $\beta_{p\mathcal{M}_n} = (\tau'_1, \tau'_2, \dots, \tau'_y)$ of MSCs $\mathcal{M}_1$, …, and $\mathcal{M}_n$ in $\mathcal{G}_p$ as $\Theta_p = (\theta_1, \dots, \theta_n)$ such that Θ_p is the set that $\Theta_p \subseteq \beta_{p\mathcal{M}_i}$ where $i \in \{1, \dots, n\}$. Then we have the following: $\Theta_p = (\theta_1, \dots, \theta_n) = (\tau_w, \dots, \tau_x) = \cdots = (\tau'_a, \dots, \tau'_b)$ and $\theta_i = e \in E'_p$.

Process p may show an unexpected behavior if at least one of the following conditions is satisfied in at least one of the MSCs $\mathcal{M}_1$, …, or $\mathcal{M}_n$.

Case I: $for\ \tau_{x+1} \in \beta_{p\mathcal{M}_1}, \dots, \tau'_{b+1} \in \beta_{p\mathcal{M}_n}\ we\ have\ (\tau_{x+1} \in S)\ and/or\ \ (\tau'_{b+1} \in S)$.

Case II: $for\ \theta_n \in \Theta_p\ we\ have\ \theta_n \in S$.

Case III:

$for\ \theta_i < \theta_z \in \Theta_p\ we\ have\ \theta_z \in S\ and\ \exists\ \mathcal{M}_x\ and\ \mathcal{M}_y\ such\ that\ \ \mu(\theta_z) = p!\, q_i(m')\ in\ \mathcal{M}_x\ and\ \mu(\theta_z) = p!\, q_j(m')\ in\ \mathcal{M}_y$ And $q_i \neq q_j$.

These conditions determine various situations that can lead to emergent behavior **CLEB-I**, which is caused by shared states of one process in two or more MSCs. Case I explains the situation that the immediate state of process p after its shared states belongs to the send events. Figure 7 shows a sample of this case. The Gold messages in this figure are the shared states for component C1 and the red one is the case explained in Case I.

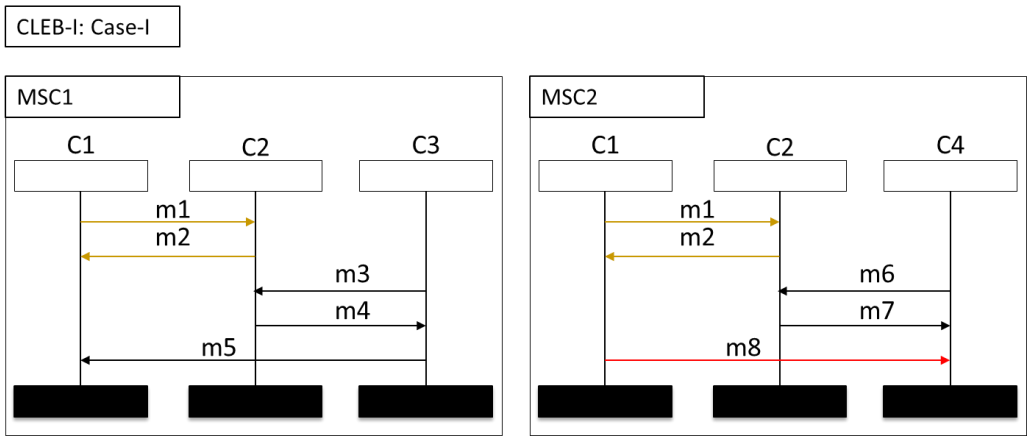

Figure 7. CLEB-I Case I

Case II defines the situation that process p has a send interaction in its last element of its shared states. Either of these situations can lead to an emergent behavior; since p can send a message in these states, resulting in a confusion of the MSC that other processes are accomplishing their functionalities. This case is shown in Figure 8. In this figure, the shared states are shown with Gold arrows for component C1. The red arrow is Case II that can cause an emergent behavior.

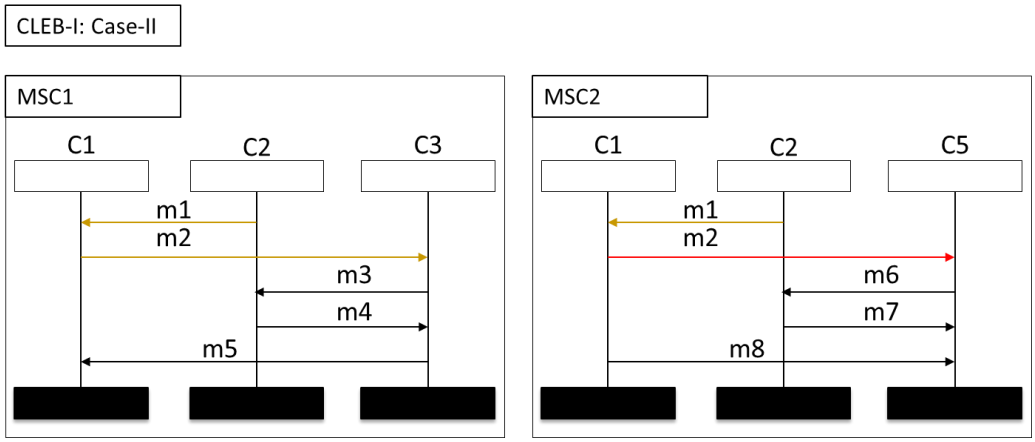

Figure 8. CLEB-I Case II

The third condition, case III, expresses a situation that process p sends the same message in one of its shared states to different processes in at least two of the MSCs. A simple set of MSCs

showing this case is represented in Figure 9. In these pictures, the Gold arrows (m1, m, and m2) are the shared states of component C1 and the red one with message label *m* is the one that is explained in Case III.

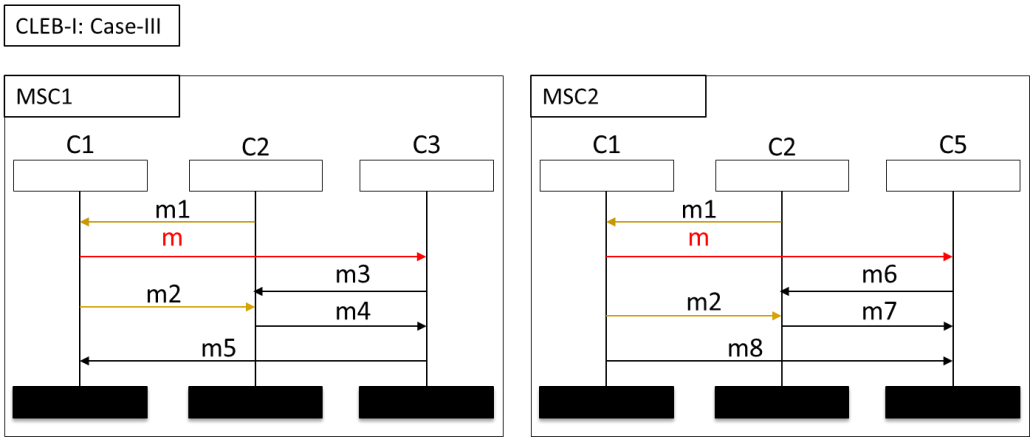

Figure 9. CLEB-I Case III

5.2.1.2 Causes

The main cause of the occurrence of CLEB-I is that the process sends a message in or after its shared states.

Consider a process p that has some shared states in some of the MSCs it is involved in. If there is no condition on sending the specific messages in the definitions, p can send this message whenever the situations are satisfied. In other words, it can send a message whenever it goes to the defined states that it can send a message. Therefore, it is probable that processes are performing their tasks in another MSC, but, because process p is active and has the option to send a message, it proceeds its actions as if it is continuing in $\mathcal{M}$. This difference, makes an unexpected behavior that p is caused in the system and we refer to it as CLEB-I.

5.2.2 CLEB-II: Respond to different components

This emergent behavior occurs when an agent p loses the information about its receiving interactions, when the same messages are sent to p by different agents in various MSCs and bring the agent into the same state (a shared state). In this situation, the agent p has some shared states in these MSCs. If p misses the information about its communications (e.g. senders or receivers of its actions) in these MSCs, it can be confused which MSC to proceed, in or after its shared states. Therefore, agent p may have started to perform its tasks in $\mathcal{M}_i$, but it can continue its actions in another MSC $\mathcal{M}_j$. This case happens if agent p is a sender of a message in some states in these two MSCs. In this case, a new scenario is implied to the system, which is caused by a confusion in actions of agent p in the two MSCs $\mathcal{M}_i$ and $\mathcal{M}_j$. We refer to this emergent behavior as **CLEB-II**.

5.2.2.1 Specification

Consider process p which is receiving some messages in MSCs $\mathcal{M}_{\mathrm{i}}$ and $\mathcal{M}_{\mathrm{j}}$ in its high level structure $\mathcal{G}_{\mathrm{p}}$. Let $\Theta_p = (\theta_1, \dots, \theta_n) = (\tau_w, \dots, \tau_x) = (\tau'_a, \dots, \tau'_b)$ be the shared states of process p in these MSCs and $\theta_i = e \in E_p$. Emergent behavior **CLEB-II can occur** if we have

Condition I: $\exists m \in M \; and \; \exists \theta_i \in R \; such \; that \; \mu(\theta_i) = p?\, q_i(m) \; in \; \mathcal{M}_i \; and \; \mu(\theta_i) = p?\, q_j(m) \; in \; \mathcal{M}_j$ And $q_i \neq q_j$.

And at least one of the following cases are satisfied:

Case I:

$for \; \tau_{x+1} \in \beta_{p\mathcal{M}_i}, \tau'_{b+1} \in \beta_{p\mathcal{M}_j} \; we \; have \; (\tau_{x+1} \in S) \; and/or \; (\tau'_{b+1} \in S)$.

Or $for \; \theta_n \in \Theta_p \; we \; have \; \theta_n \in S$.

Case II: $for\ \theta_i < \theta_z \in \Theta_p\ we\ have\ \theta_z \in S\ and\ \mu(\theta_z) = p!\,q_i(m')\ in\ \mathcal{M}_i\ and\ \mu(\theta_z) = p!\,q_j(m')\ in\ \mathcal{M}_j$.

Case III: $for\ \theta_i < \theta_z \in \Theta_p\ we\ have\ \theta_z \in S\ and\ \mu(\theta_z) = p!\,q_x(m')\ in\ \mathcal{M}_i\ and\ \mu(\theta_z) = p!\,q_y(m')\ in\ \mathcal{M}_j$ And $q_x \neq q_y$.

The **Condition I** defines that, there is a shared state for process p in the two MSCs, in which, p has same interactions of "receive" type with different processes q_i and q_j. Both communications cause process p to go into same states. In **Case I**, we refer to a situation that the last shared state θ_n of process p, or its immediate next state, in one or both MSCs, are a send event. This case is shown in Figure 10. Similar to the previous figures, the Gold arrows show the shared states and the red ones are the specific conditions defined in this case. The red message in MSC1 shows the case that the last shared state is of *Send* type, and the red one in MSC2 demonstrates the case when the immediate next state after the shared states is of type *Send*.

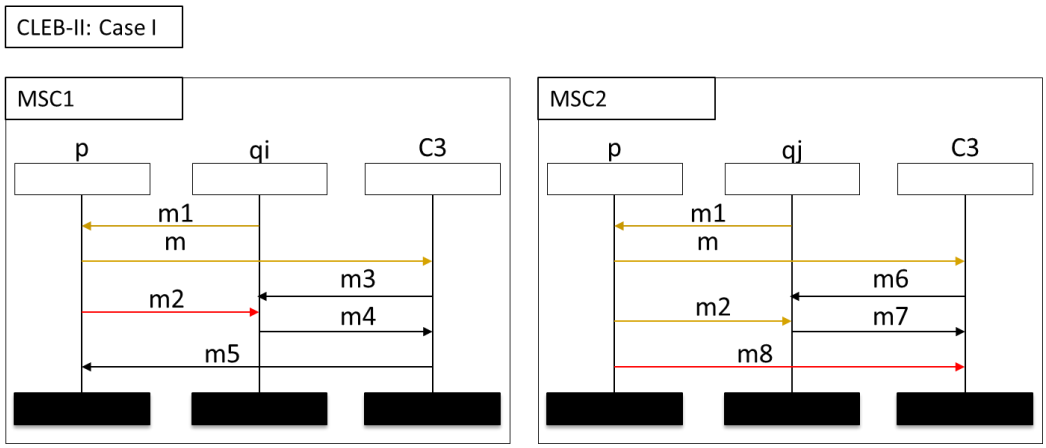

Figure 10. CLEB-II Case I

Case II defines that if there is a send event in the shared states of process p in the two MSCs (θ_z) that comes after the state found in Condition I (θ_i), and the receivers are processes q_i and q_j

(q_i and q_j are the same processes in condition I), an implied scenario **CLEB-II** can exist. Processes q_i and q_j are the senders of same messages to process p that brings process p in its shares states, in MSCs $\mathcal{M}_\text{i}$ and $\mathcal{M}_\text{j}$, respectively. This send event from process p, is mostly in response to the same message that it has received from processes q_i and q_j. We have illustrated this case in Figure 11. The red arrow represents the state explained in this case.

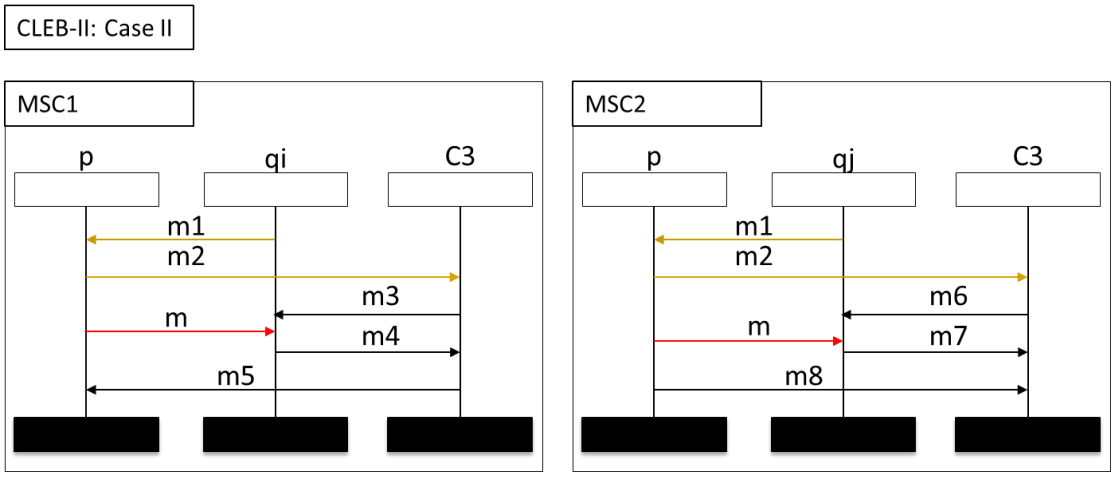

Figure 11. CLEB-II Case II

Finally, **Case III** specifies a general situation of **Case II**. If there is a send state θ_z in Θ_p that comes after the state θ_i (found in Condition I) and the receivers are processes that are different from those who brought p into its shared states (processes q_i and q_j), an implied scenario can occur. In **Case II**, we refer to a situation that agent p confuses when sending a message to q_i and q_j, as a response to the request/messages of its previous interactions. However, **Case III** specifies that even when p sends a message to two other processes, other than q_i and q_j that have sent the same messages to p, it still may show an emergent behavior. This is shown in Figure 12.

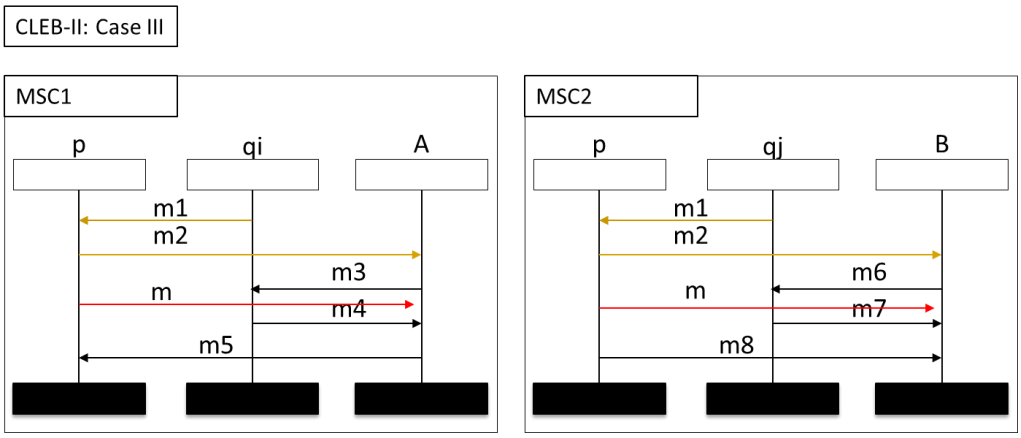

Figure 12. CLEB-II Case III

5.2.2.2 Causes

In CLEB-II, the reason of having an implied scenario exists in the previous shared states of process p. Since process p is responding to one of its previous interactions (received messages), it should have the information about the sender, or a track to the sender of a message. However, if process p does not have this information, it may respond to some processes other than the ones it is supposed to send a message, in or after its shared states. This is not in accordance to any of the MSCs, or the MSC that the rest of processes are following. Consequently, it can cause an emergent behavior in the system.

5.2.3 CLEB-III: Local branching choices

The emergent behavior CLEB-III is a local branching choice, in which one process has the option to choose between two scenarios of the system. For this type of emergent behavior, we investigate the high level structure $\mathcal{G}_p$ of a process in the whole system, by analyzing its behavior in the hMSC. If the process is an active process $p_{a\mathcal{M}}$ (sends the first message in the scenario, without waiting for other processes to change its states (by receiving a message)) in various branches of an hMSC, it should follow certain conditions applied to the starting states in each branch in order

to prevent an emergent behavior. Also, the presence of other active processes in these SDs can lead to SLIS-III. In addition, if the process has shared states in these SDs, it should be considered for CLEB-I as well. This case is referred to as *local* branching choice, since only one process triggers the choice in the branch of the hMSC.

5.2.3.1 Specification

Consider the hMSC of the system $\mathcal{G} = (P, M, \boldsymbol{\mathcal{M}}, V, Ed, \mathsf{C}, F_0, F_f)$ that contains two branches $\mathcal{G}_i = (M_i, \boldsymbol{\mathcal{M}}_i, V_i, Ed_i, \mathsf{C}_i, F_{0_i}, F_{f_i})$ and $\mathcal{G}_j = (M_j, \boldsymbol{\mathcal{M}}_j, V_j, Ed_j, \mathsf{C}_j, F_{0_j}, F_{f_j})$.

Let process $p \in P$ in hMSC has high level structure $\mathcal{G}_p = (M_p, \boldsymbol{\mathcal{M}}_p, V_p, Ed_p, \mathsf{C}_p, F_{0_p}, F_{f_p})$.

The following conditions should be satisfied that CLEB-III occurs:

When considering high level structure $\mathcal{G}_p$, process p has interactions in both branches $\mathcal{G}_\mathrm{i}$ and $\mathcal{G}_\mathrm{j}$.

Process p be an active process in at least one of the branches $\mathcal{G}_\mathrm{i}$ and $\mathcal{G}_\mathrm{j}$ ($\boldsymbol{\mathcal{M}}_i \; and/or \; \boldsymbol{\mathcal{M}}_j \in \boldsymbol{\mathcal{M}}_{pa} \subseteq \boldsymbol{\mathcal{M}}$).

If the process is an active process in at least one of its branches, it may follow a different branch, while it is supposed to execute another branch with other processes in the system. On the other hand, if the process is always a receiver process (passive) in these branches, there is no chance that it can cause an emergent behavior by its behavior.

The general case of CLEB-III is when we have more than two branches in the high level execution of the scenarios of the system. An example of this CLEB-III is demonstrated in Figure 13. At the left side of the figure, a sample high level structure (hMSC) of the system is shown. The branch in the system is a choice between MSCs M2 and M3. As it is shown in the right side of the figure, process p has functionalities in both M2 and M3. However, in M2, it acts as an active

process. Therefore, when the system comes to the branch, there is a chance that other processes are executing MSC3, while process p can start with sending a message in MSC2, if there is no condition for it to be satisfied. This is shown in red in this figure.

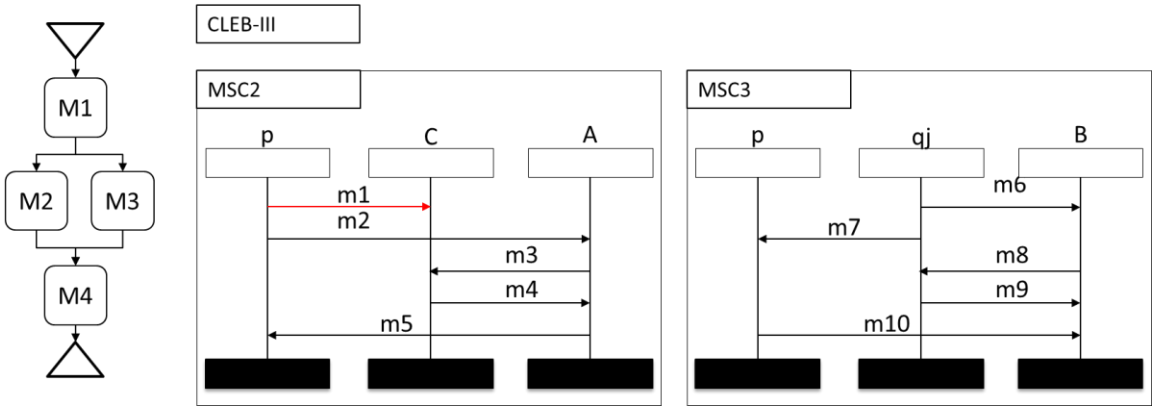

Figure 13. CLEB-III

5.2.3.2 Causes

If the process is an active process in at least one of its branches – it is active in at least one of the MSCs that make a branch in its high level structure – it may follow a different branch, while it is supposed to execute another branch with other processes in the system. This happens because when the process is active, and there is no restriction on the conditions it can start its functionalities in the next MSC, it will not wait for other processes and it is not aware of the state of the whole system. Therefore, it can choose between any of the branches, which can cause an emergent behavior. On the other hand, if the process is always a receiver process (passive) in these branches, there is no chance that its behavior causes an emergent behavior of type CLEB-III, since it does not have the option to continue its actions. To avoid this emergent behavior, the process should follow some conditions on the MSCs in which it acts as an active process.

5.2.4 CLEB-IV: Race conditions

Race condition in our research field is referred to as accessing or sending a message by various processes, while the order of the receipt of the message is important for the receiver process. It can cause a change in the functionalities/behaviors of the receiver process based on the received message [9]. In literature, race condition is defined as a condition in which the order of the events is not guaranteed in practice [152].

Suppose that processes $p, q, r \in P$ have interactions with each other in an MSC $\mathcal{M}$. If process p has at least one receive message from each of the other processes q and r, and there is no control over the order of receive messages for process p with respect to its specification, a race condition can occur. In this situation the order of the messages and the agents' interactions that were specified in the system specifications in MSC $\mathcal{M}$ is not preserved and therefore it can result in an EB. However, not all of the changes in the received order of messages can result in CLEB-IV. The first situation is when there is a change in the order of receiving messages for process p which is not in the events set of process p in the set of MSCs of this process. If this situation is satisfied, CLEB-IV can occur.

The other case is more restrictive and is related to the behavior of process p and its interactions with other agents. In this case, the agents that process p interacts with should also be preserved in the change of the order of its received messages, in order to make sure that p has no EB of type CLEB-IV.

5.2.4.1 Specification

Suppose that processes $p, q, r \in P$ have interactions with each other in an MSC $\mathcal{M}$. If process p has at least one receive message from each of the other processes q and r, and there is no control over the order of receive messages for process p with respect to its visual order $<_p$, a race condition

can occur. In this situation the visual order $<_p$ in MSC $\mathcal{M}$ is not preserved. There are two cases in the following that specify this situation. If either one of these cases are satisfied, an implied scenario CLEB-IV can occur.

Case I: Let $\gamma_{p\mathcal{M}} = (b_1, \dots, b_r, \dots, b_q, \dots, b_n)$ be the receive vector of process p in an MSC $\mathcal{M}$. The elements b_r and b_q represent the set of events that p receives from processes r and q respectively. Let $\gamma_{p\mathcal{M}}{}' = (b_1, \dots, b_q, \dots, b_r, \dots, b_n)$ be the new receive vector for process p, where the order of b_r and b_q has changed, compared to $\gamma_{p\mathcal{M}}$. We have $\beta_{p\mathcal{M}} = (\cup\left(\delta_{p\mathcal{M}}, \gamma_{p\mathcal{M}}\right), <_p)$ as the state vector of p and $\beta_{p\mathcal{M}}{}' = (\cup\left(\delta_{p\mathcal{M}}, \gamma_{p\mathcal{M}}{}'\right), <_p)$ as the new state vector of process p.

If $\beta_{p\mathcal{M}}{}'$ is not in the set of $\beta_p = \cup_{\mathcal{M}} (\beta_{p\mathcal{M}})$, then process p can cause CLEB-IV because of a change in its receive orders:

$$if\ \exists m, m' \in M;\ e, e' \in E_p; \mu(e), \mu(e') \in \textstyle\sum_p\ such\ that\ \mu(e) = p?\, q(m)\ and\ \mu(e')$$

$$= p?\, r(m')\ in\ \mathcal{M}'\ and \neg\left(\beta_{p\mathcal{M}}{}' \subseteq \beta_p\right)$$

$$then\ \beta_{p\mathcal{M}}{}' \text{ is an emergent behavior } X \text{ for } p.$$

This case indicates that a change in the order of receiving messages for process p should be in the events of process p in other MSCs, or receiving of such messages should be controlled, in order to prevent an emergent behavior. This is presented in Figure 14. The left side of the figure shows MSC M which is in the system specifications. In the right hand side, there is an MSC that is implied to the system because of a change in the order of the message that are received by process P. The only case that may prevent having such an MSC is to have MSC implied in the specifications of the system.

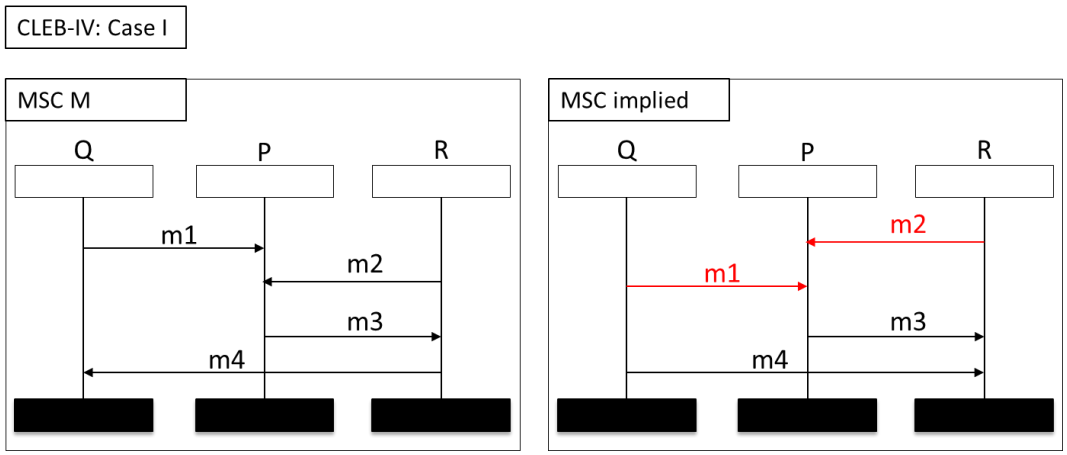

Figure 14. CLEB-IV Case I

Case II: Let $\gamma_{p\mathcal{M}}{}' = (b_1, \dots, b_q, \dots, b_r, \dots, b_n)$ be a new receive vector for process p, where the order of b_r and b_q has changed, compared to $\gamma_{p\mathcal{M}}$ and $\exists \mathcal{M}' \in \boldsymbol{\mathcal{M}}$ such that $\beta_{p\mathcal{M}}{}' \subseteq \beta_p$. Let $G_{p\mathcal{M}} = (V_{p\mathcal{M}}, \boldsymbol{E}_{p\mathcal{M}}, \boldsymbol{f})$ be the associated interaction graph for receive vector $\gamma_{p\mathcal{M}}$ and state vector $\beta_{p\mathcal{M}}$ in $\mathcal{M}$. Consider $G_{p\mathcal{M}}{}' = (V_{p\mathcal{M}}{}', \boldsymbol{E}_{p\mathcal{M}}{}', \boldsymbol{f}')$ as the associated interaction graph for receive vector $\gamma_{p\mathcal{M}}{}'$ and state vector $\beta_{p\mathcal{M}}{}'$ in $\mathcal{M}'$. We define $n_q{}', n_r{}' \in N_{p\mathcal{M}}{}'$ as the corresponding nodes in $G_{p\mathcal{M}}{}'$, for the states of process p that receive messages from processes q and r, respectively. If $G_{p\mathcal{M}}{}'$ is not in the set of interaction graphs of process p, $G_p = \cup\, G_{p\mathcal{M}}$, then $\beta_{p\mathcal{M}}{}' = (\cup\, (\delta_{p\mathcal{M}}, \gamma_{p\mathcal{M}}{}'), <_p)$ is an emergent behavior CLEB-IV.

Case II defines more restrictive criteria that a process can have a behavior that leads to implied scenario CLEB-IV. Other than the new receive vector in the states of process p, the state transition vectors should be considered as well; to check whether or not a process can lead to CLEB-IV and show an emergent behavior.

In these cases, a change in the order of states b and b', resulted from communications with processes q and r, in the receive states of process p, creates a new receive vector $\gamma_{p\mathcal{M}}{}'$. The $\gamma_{p\mathcal{M}}{}'$

can exist in the state vectors of p in another MSC $\mathcal{M}'$. However, the communicating processes associated to states b and b' in $\mathcal{M}'$, may be different, namely, processes y and z. Therefore, the two state vectors $\beta_{p\mathcal{M}}$ and $\beta_{p\mathcal{M}}{}'$, are not considered as similar state vectors when considering their associated state transition vectors $\varphi_{p\mathcal{M}}$ and $\varphi'_{p\mathcal{M}}$; because their communicating processes for these two states (b and b') are different. In this situation, although the state vectors in the two MSCs are the same, process p can still show an emergent behavior, because of a difference in its state transition vectors. These conditions are shown in Figure 15. The two MSCs M1 and M2 are the legal MSCs of the system. M1 represents the receipt of messages m1 and m2 of process P from processes Q and R respectively. If the order of the receipt of the message m1 and m2 changes and this new receive vector exists in the legal vectors of process P, we still cannot be sure about not having an implied scenario. As it is shown, this change in the order of receipt of m1 and m2 is allowed in M2. However, the senders are processes Y and Z which are different from Q and R. Therefore, this change in the order of receiving of messages can cause an implied scenario, as it is shown in MSC implied in Figure 15. These changes are shown in red in this MSC.

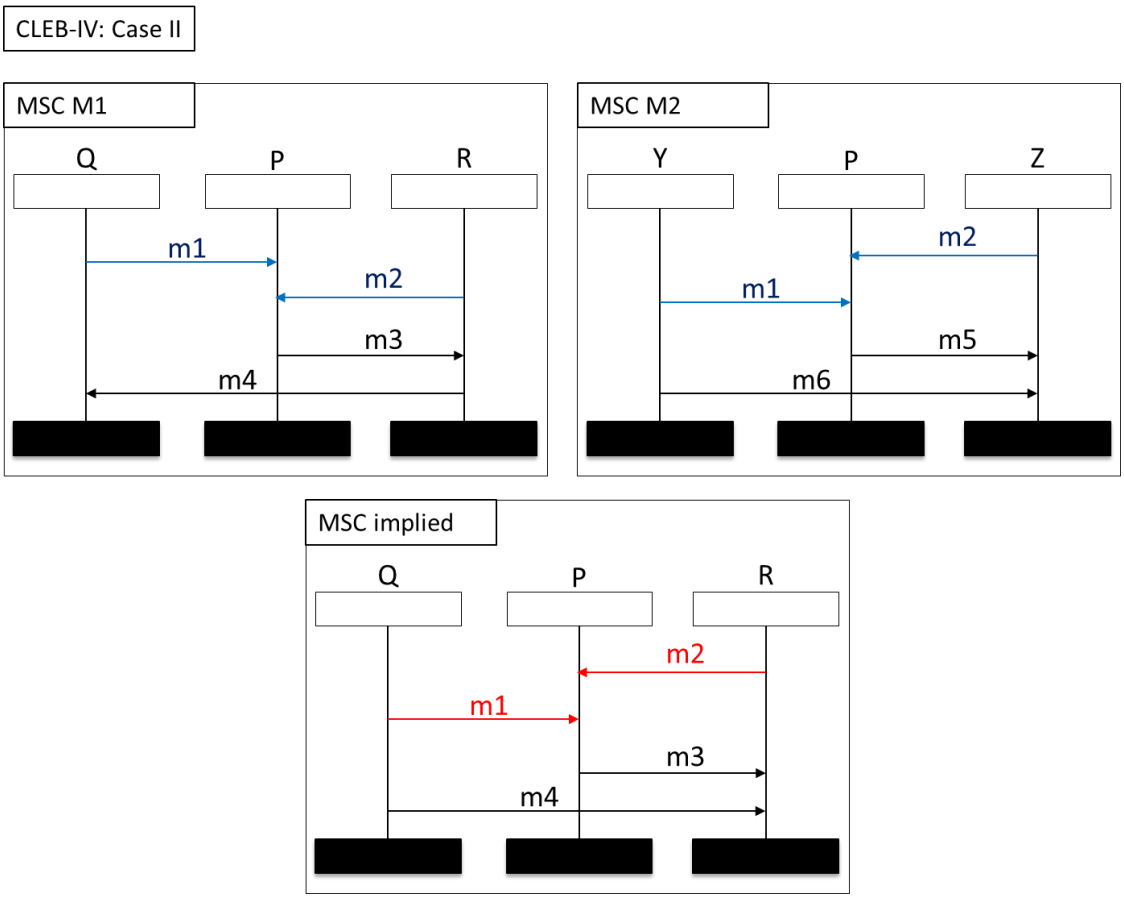

Figure 15. CLEB-IV Case II

5.2.4.2 Causes

The change in the received messages of one process can change its behavior, since it is acting based on the received events. Therefore, the changes in the received events should be controlled to certify if the change of events on a process causes its behavior acceptable or not. This change is caused by a race condition. A race condition can be interpreted as the race between senders to send a message to the receiver agent. For example, in a web service application, there can be a race between various agents to send requests to the server agent. The result will change based on the messages received by the agent, since it will follow the next interactions based on the events it has received. However, not always care is taken to include these situations. For example refer to a case

of intelligent agents for traffic and congestion control. A race condition in this case can increase the heavy traffic problem and can result in false changes in traffic lights.

5.3 System level implied scenarios

The system level implied scenario (SLIS) class is divided into four sub-classes in order to specify the conditions that must be satisfied for the occurrence of each type. The four classes are:

1. SLIS-I: Shared interactions
2. SLIS -II: Behavior combination
3. SLIS -III: Non-local branching choice
4. SLIS -IV: Asynchronous concatenation

In the first class, some interactions between a number of components are the same and repeated in different scenarios of the system. In the second class, the behavior of various components from different MSCs can be combined into one MSC, which does not exist. In other words, in the execution time, the components may accomplish their functionalities from different MSCs. Consequently, a new scenario can be implied to the system. The third class specifies a case in which there is a branch in the hMSC of the system. The processes that can decide on which branch to follow next, are not the same. This can cause an SLIS in the system. Finally, the asynchronous concatenation of MSCs can bring an issue to the system, in which components will not wait for other components to finish their functionality in one MSC. Since there is no blocking or wait function, the components may executing different MSCs, although they have started the same MSC at the same time. The specification and details of each class are presented in this chapter.

5.3.1 SLIS-I: Shared interactions

This type of implied scenario can be considered as an extension to CLEB-I, since it deals with the shared states and interactions between components. In this class, there might be some shared interactions (shared states of some components that are communicating through the same states in various scenarios) between two or more components in various scenarios of the system. These interactions can be considered as a part of an interaction graph that is shared in various interaction graphs of the system. If the behavior of at least one agent in these interactions is nondeterministic, an implied scenario may occur.

5.3.1.1 Specification

To specify SLIS-I, the following definitions are required.

Definition 1. Joint shared states Θ_{pq} (shared states between two processes in MSCs $\mathcal{M}_{Mut}$)

Suppose that we have shared states vector $\Theta_p = (\theta_1, \dots, \theta_n)$ for process p and $\Theta_q = (\theta'_1, \dots, \theta'_m)$ as the shared states vector of process q in $\mathcal{M}_{Mut}$ MSCs. We define the ***joint shared states*** Θ_{pq} for two processes p and q in a set of MSCs $\mathcal{M}_{Mut}$ as the set of their interactions that appears in all MSCs of $\mathcal{M}_{Mut}$ and follows the same visual order $((\cup \{<_p, <_q) \cup C)$ as the visual orders of processes p and q in these MSCs. The joint shared states is $\Theta_{pq} = \{P_{pq}, \mathcal{M}_{Mut}, \Theta\Theta, C_{pq}\}$, where:

P_{pq} shows the set of interacting processes in Θ_{pq} (p and q).

$\mathcal{M}_{Mut}$ is the set of MSCs that p and q have mutual shared states.

$\Theta\Theta = \{\Theta_p, \Theta_q\}$ is the set of shared states of p and q in MSCs $\mathcal{M}_{Mut}$. $\forall \theta_i \in \Theta_p$ we have $\mu(e) = p!\, q(m_i)$ *or* $\mu(e) = p?\, q(m_i)$, and $\forall \theta'_i \in \Theta_q$ we have $\mu(e) = q?\, p(m_i)$ *or* $\mu(e) = q!\, p(m_i)$.

C_{pq} represents the edges (interactions) in the MSCs between processes p and q with respect to their visual orders.

Implied scenario **SLIS-I** can occur when two processes p and q have joint shared states $\Theta_{pq} == \{P_{pq}, \mathcal{M}_{Mut}, \Theta\Theta, C_{pq}\}$ in MSCs $\mathcal{M}_{Mut}$ and the following condition is satisfied:

$\exists z \in P_{pq}\ and\ \exists \mathcal{M} \in \mathcal{M}_{Mut}\ such\ that\ for\ \theta_n = \tau_i\ and\ \theta_n \in \Theta_z\ , \tau_{i+1} \in S\ and\ \ \tau_i, \tau_{i+1} \in \beta_z.$

This condition specifies that the joint shared states of two processes may lead to an implied scenario, if at least one of the processes sends a message in one of the MSCs in $\mathcal{M}_{Mut}$ in the next state of its joint shared states. Unless the next state of its joint states is part of the component's shared state and is always sent to one specific process.

We define the mutual shared states between a set of processes as the general form of the joint states for two processes as follows.

Definition 2. Mutual shared communications Com_{Mut}

The ***mutual shared communications*** $Com_{Mut} = \{P_{Mut}, \mathcal{M}_{Mut}, \Theta\Theta_{Mut}, C_{Mut}\}$ is defined for processes $P_{Mut} = \{p_1, \dots, p_x\}$ that are interacting in MSCs $\mathcal{M}_{Mut}$. The C_{Mut} is the set of continuing interactions such that it appears in the same visual order and between the same processes in all MSCs of $\mathcal{M}_{Mut}$ and that use the processes' states in $\Theta\Theta_{Mut}$. The $\Theta\Theta_{Mut}$ is the set of mutual shares states of processes in P_{Mut} in MSCs $\mathcal{M}_{Mut}$.

Definition 3. Mutual shared states $\Theta\Theta_{Mut}$

The ***mutual shared states*** $\Theta\Theta_{Mut} = \{\Theta_{p_1}, \dots, \Theta_{p_x}\}$ is defined as a set of shared states of processes in $P_{Mut} = \{p_1, \dots, p_x\}$ over MSCs $\mathcal{M}_{Mut}$ that have mutual shared communications. All

of these states follow continuous interactions of processes P_{Mut} over the C_{Mut} edges in each of the MSCs in $\mathcal{M}_{Mut}$.

Implied scenario **SLIS-I** can occur when a set of processes P_{Mut} have mutual shared states $\Theta_{Mut} = \{P_{Mut}, \mathcal{M}_{Mut}, \Theta\Theta_{Mut}, C_{Mut}\}$ in MSCs $\mathcal{M}_{Mut}$ if the following condition is satisfied:

$$\exists z \in P_{Mut} \; and \; \exists \mathcal{M} \in \mathcal{M}_{Mut} \; such \; that \; for \; \theta_n = \tau_i \; and \; \theta_n \in \Theta_z \, , \tau_{i+1} \in S$$

$$and \;\; \tau_i, \tau_{i+1} \in \beta_z$$

This condition specifies that there should be at least one process in P_{Mut} that sends a message in one of the MSCs in $\mathcal{M}_{Mut}$ in the next state of its mutual shared states.

The concept of shared interactions is shown in Figure 16. Messages m1-m3 are sent and received by components C1 and C2 in the two MSCs in this figure. These are considered as shared interactions. The conditions that can cause an implied scenario is the existence of a send message by either one of C1 or C2 in these shared interactions. This will be m3 in this example.

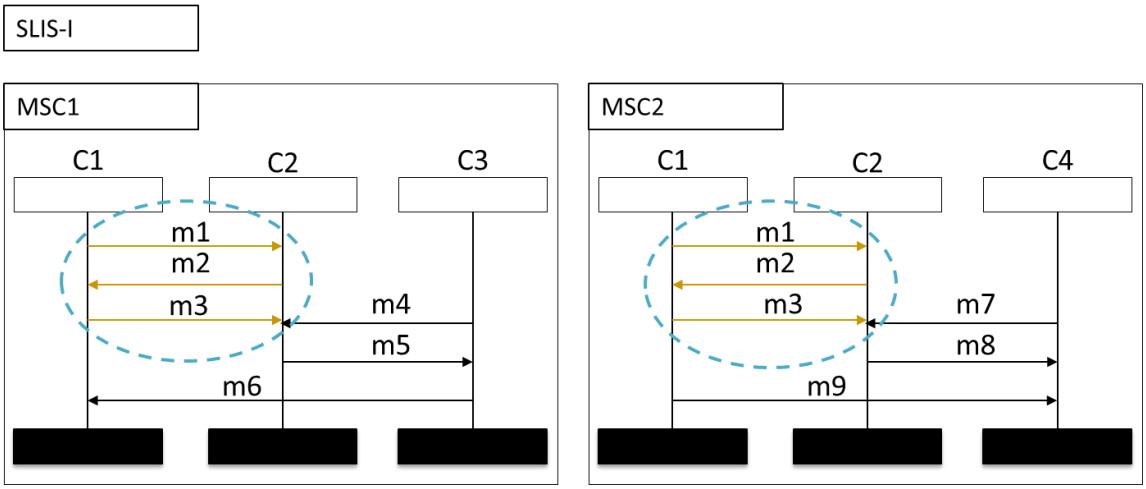

Figure 16. SLIS-I shared interactions

5.3.1.2 Causes

In SLIS-I, the implied scenario occurs because one or more processes can send a message without restriction in or after their last shared state. The reasons of the occurrence of this type of implied scenario are similar to the ones for the CLEB-I.

5.3.2 SLIS-II: Behavior combination

For definition of this type of implied scenario, we refer to [88, 95, 97, 98] paper. In **SLIS-II**, the implied scenario is caused by the combination of behaviors of two or more processes, from various MSCs of the system into one MSC. The new scenario is a collection of the behavior of different processes in different MSCs. In other words, in this new scenario, one process shows the behavior specified for it in one MSC, and another process performs its defined tasks of another MSC. The result is an implied scenario, which is not in conformance with any of the MSCs of the system. This implied scenario is the result of having two or more active processes in various MSCs of the system that do not have any restrictions on their first actions. Various cases may exist that can result in different issues in the system. Specific conditions must exist that it leads to an implied scenario. The details are given below. It is worth mentioning that for the detection of this implied scenario no hMSC is required.

5.3.2.1 Specification

Consider that processes p and q are active processes in MSC $\mathcal{M}_{\mathrm{x}}$ and MSC $\mathcal{M}_{\mathrm{y}}$ respectively. For the occurrence of SLIS-II, there should be at least one active process in each of the two MSCs, which are defined as below.

Condition I.

The first action of process p ($p_{a\mathcal{M}_{\mathrm{x}}}$) is: $\exists m \in M |\, \mu(e) = p_{a\mathcal{M}_1}!\, p_i(m)$.

The first action of process q ($q_{a\mathcal{M}_{\mathrm{y}}}$) is: $\exists m' \in M |\, \mu(e') = q_{a\mathcal{M}_2}!\, p_j(m')$.

There is no condition on taking the first action of p in $\mathcal{M}_{\mathrm{x}}$ ($\mu(e)$ for $p_{a\mathcal{M}_{\mathrm{x}}}$ in $\mathcal{M}_{\mathrm{x}}$) or there is no condition on taking the first action of q in $\mathcal{M}_{\mathrm{y}}$ ($\mu(e')$ for $q_{a\mathcal{M}_y}$ in $\mathcal{M}_{\mathrm{y}}$).

In implied scenario **SLIS-II**, there is a new scenario $\mathcal{M}_{\mathrm{new}}$, in which, both $\mu(e) = p_{a\mathcal{M}}!\,p_i(m)$ and $\mu(e') = q_{a\mathcal{M}}!\,p_j(m')$ exist. Some conditions must exist that SLIS-II occurs. In the following, we determine these cases and explain various situations based on the actions of receiver processes of p and q and whether the receiver processes are the same process in these scenarios. The implied scenario SLIS-II is shown in Figure 17. The two MSCs in the above are the cases that P and Q are sending messages to two other processes. The MSC in the bottom of the figure is the implied scenario caused by the combination of the behavior of the two other MSCs, in which the first actions of P and Q occur in one MSC.

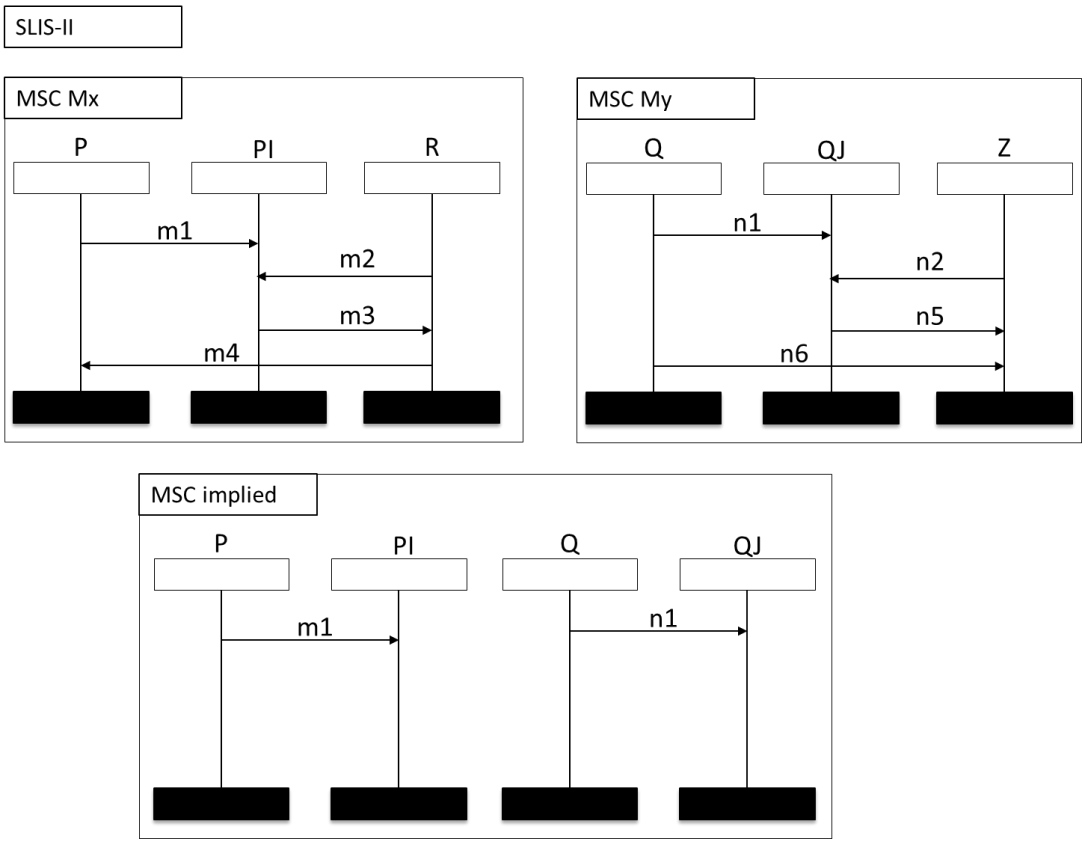

Figure 17. SLIS-II

5.3.2.1.1 Cases

Suppose the receivers of the first messages that processes p and q send in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$ are processes p_i and p_j, respectively. We investigate two cases regarding the interactions of p_i and p_j in these scenarios. For each of these cases, we also determine three situations that is based on these criteria: p_i and p_j are same processes, are different processes, or are processes q and p.

5.3.2.1.1.1 Case I

In this case, the receiver processes (p_i and p_j) do not send any other messages in MSCs $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$. In other words, p_i and p_j do not have any other interaction of type Send in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$ respectively. In this case, since the receiver processes do not have any other interactions, they do not interact based on the information they have received. Therefore, the problem stays in the receiver processes and is not propagated into the other processes of the system. However, if the information requires an update for the receivers, the receiver processes may not be updated correctly and may work with the wrong information in the MSCs of the system. Implied scenario SLIS-II may happen or cannot occur in each of the following situations:

Case I-1: $p_i, p_j \neq q, p \; and \; p_i \neq p_j$

In this case, the processes p, q, p_i, and p_j are four different processes. If condition I is satisfied, an implied scenario can occur, unless there is an MSC in the system that has the interactions among all processes in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$ and with the same visual order (i.e. interactions of p_i and p, interactions of p_j and q, and all other interactions that are in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$). Figure 18 illustrates the explanation of this case.

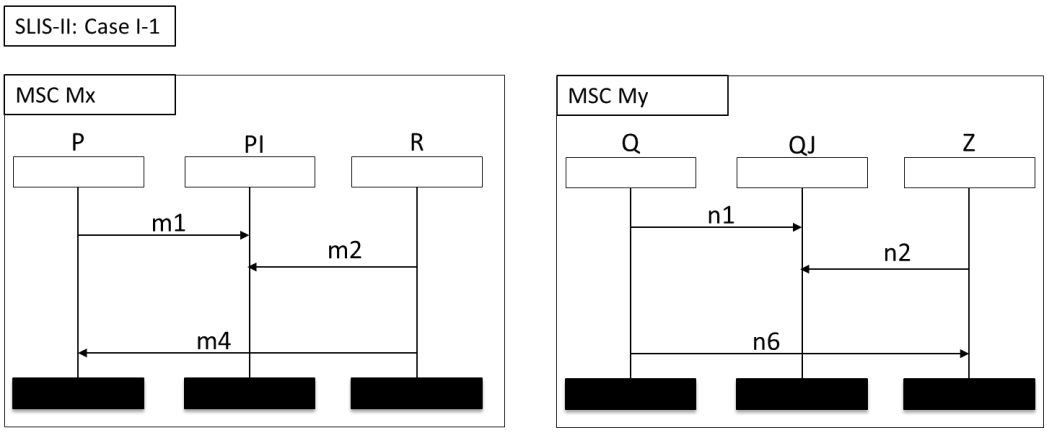

Figure 18. SLIS-II Case I-1

Case I-2: $p_i = q \; and \; p_j = p$

This situation specifies that the processes p and q are sending messages to each other. In this case, either of the messages is sent and received earlier, prevents the other active process to send a message; since in the MSCs, the receivers do not act any more, and just have the role of receiving some information. Therefore, if the send actions of the active processes are also the first action in the related MSCs that should be executed by all of the processes in the system, no implied scenario happens; but the reverse is not true. This can be interpreted as a race condition. However, until we have a scenario in the system that includes the interactions of the first message sent (either from p or q) and the other interactions of other processes in the system, implied scenario SLIS-II will not occur. However, this may cause a problem in the system in terms of the required scenario that must be executed at a specific time frame. This case is analyzed in other sections. This is shown in Figure 19.

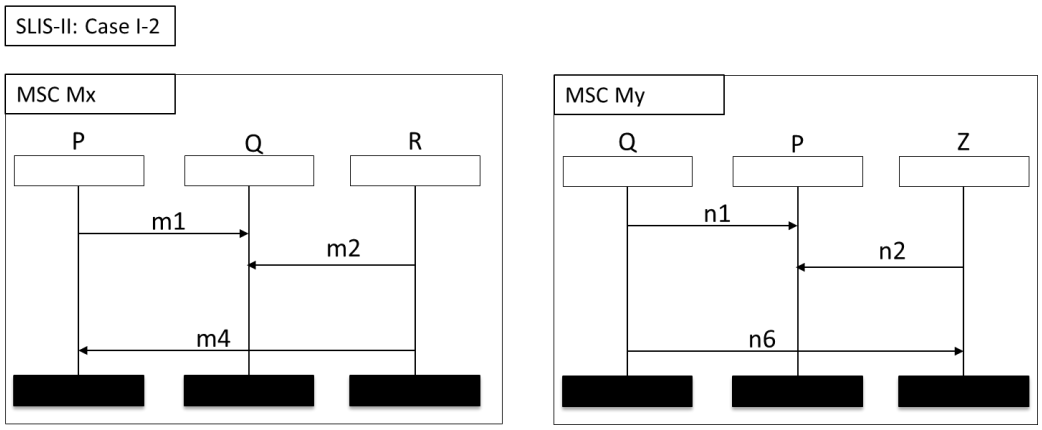

Figure 19. SLIS-II Case I-2

Case I-3: $p_i, p_j \neq p, q \; and \; p_i = p_j = w$

This case, demonstrates a situation, in which, the receivers of the messages that p and q send, in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$, are the same. This explanation is illustrated in Figure 20.

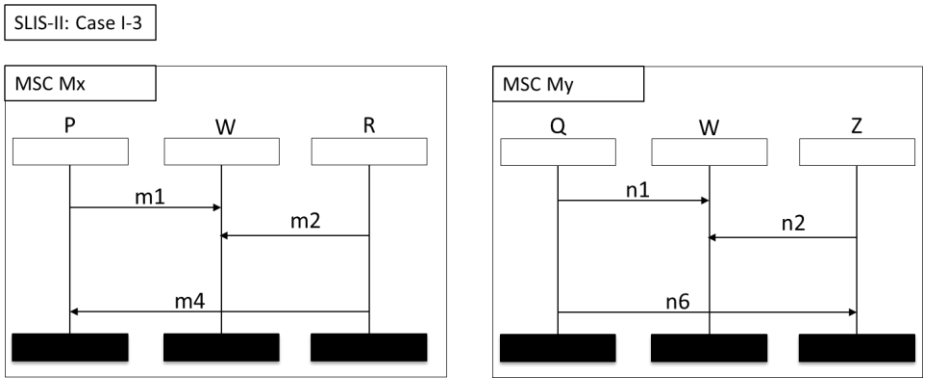

Figure 20. SLIS-II Case I-3

This explains a race condition in the system. The result is, the information that w receives, is updated, either by p or q, depending on the time of the arrival of these messages. Therefore, process w can work with wrong data in other MSCs, while it is supposed to have updated, correct information. The only condition that causes the prevention of an implied scenario SLIS-II in this case is the existence of scenarios in the system that include the ordered interactions of p and q and

the other functionalities of the rest of processes. Also, the other processes should executing the required MSC at that time. This requires four more scenarios, since it is a race condition:

1. Interactions of p to w, and then q to w, following the rest of functionalities in $\mathcal{M}_{\mathrm{x}}$;
2. Interactions of p to w, and then q to w, following the rest of functionalities in $\mathcal{M}_{\mathrm{y}}$;
3. Interactions of q to w, and then p to w, following the rest of functionalities in $\mathcal{M}_{y}$;
4. Interactions of q to w, and then p to w, following the rest of functionalities in $\mathcal{M}_{x}$;

5.3.2.1.1.2 Case II

In this case, at least one of the receiver processes (p_i and p_j) have at least one message of type send in MSCs $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$, after they have received a message from p and q, respectively. The receiver processes continue their tasks in the MSCs by sending or receiving some other messages. Consequently, the unexpected behavior of their sender processes in the implied scenario, causes the wrong information or data to be propagated to other processes in the system. It may result in major security issues, information updates, sending wrong messages to other processes, unexpected interactions with other processes, carry on wrong data to other MSCs of the system, and etc. We have discussed each problem in detail in the following. This general case is shown in Figure 21. Since the sub sections for these cases are the same, we will not show separate figure for Case II-1, Case II-2, and Case II-3.

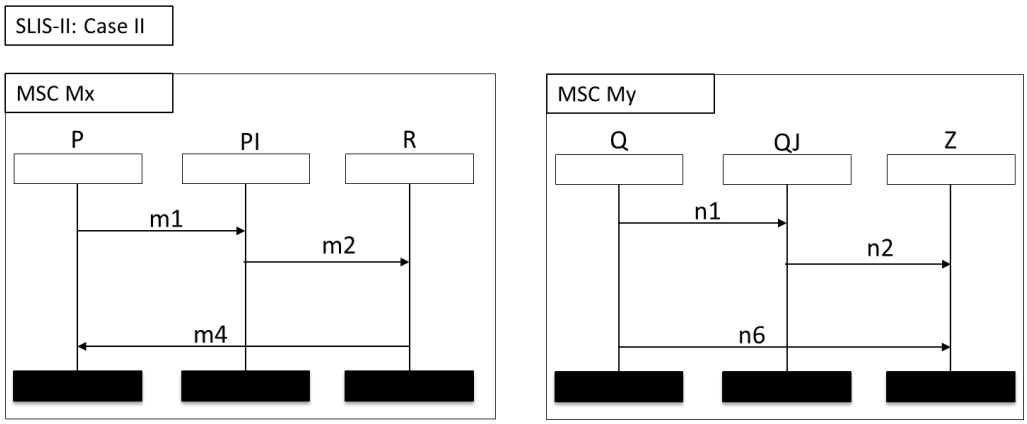

Figure 21. SLIS-II Case II

Case II-1: $p_i, p_j \neq q, p \; and \; p_i \neq p_j$

In this case, the processes p, q, p_i, and p_j are four different processes. If condition I is satisfied, an implied scenario can occur, unless there is an MSC in the system that has the interactions among all processes in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$ and with the same visual order.

Case II-2: $p_i = q, p_j = p$

This situation specifies that the processes p and q are sending messages to each other. Therefore, the receiver process is confused with its next actions. For example, process p should act based on its defined tasks in $\mathcal{M}_{\mathrm{x}}$, because it had sent a message to process q. However, as the receiver of $\mu(e') = q_{a\mathcal{M}_2}! \, p_j(m')$, process p should continue in $\mathcal{M}_{\mathrm{y}}$ or in another scenario $\mathcal{M}_i$. Therefore, the behavior of process p becomes nondeterministic. The same is applied for process q. The only situation that prevents the occurrence of SLIS-II is the existence of such a scenario that includes all the interactions for each of processes p or q; because it prevents a nondeterministic action of each of these processes.

Case II-3: $p_i, p_j \neq q, p \; and \; p_i = p_j = w$

This case, demonstrates a situation, in which, the receivers of the messages that p and q send, in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$, are the same. Consider two state vectors of process w $\beta_{w\mathcal{M}_x} = (\tau_1, \tau_2, \dots, \tau_z)$ and $\beta_{w\mathcal{M}_y} = (\tau'_1, \tau'_2, \dots, \tau'_y)$ in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$ in $\mathcal{G}_w$ respectively. State τ_i is the state in which w receives a message from p in $\mathcal{M}_x$ and state τ'_j as the state in which w receives a message from q in $\mathcal{M}_y$.

If process w does not have shared states in $\mathcal{M}_{\mathrm{x}}$ and $\mathcal{M}_{\mathrm{y}}$ after it receives messages from p and q, it confuses between its interactions in these MSCs. Since, based on the defined behaviors in $\mathcal{M}_{\mathrm{x}}$ it should take some actions, while these are different tasks from the ones that it should perform in $\mathcal{M}_{\mathrm{y}}$. Therefore, process w may show an unexpected behavior.

On the other hand, if w has some shares states Θ_w in these two MSCs immediately after τ_i in $\mathcal{M}_{\mathrm{x}}$ and after τ'_j in $\mathcal{M}_y$, $\Theta_w = (\theta_1, \dots, \theta_n) = (\tau_{i+1}, \dots, \tau_x) = (\tau'_{j+1}, \dots, \tau'_b)$, process w will not show an unexpected behavior in these states. Yet, process w should be checked for possible emergent behaviors of other types (such as CLEB-I).

5.3.2.1.2 Extensions

We define some extensions for the cases explained in the previous section.

Extension **I:** $p = q$

This case explains a situation that the same process is an active process in more than one MSC of the system. This can cause another type of implied scenario which is studied in next sections.

Extension II:

Processes p and q are active in more than two scenarios of the system. In this situation, the above mentioned cases are still valid. The only difference is in investigating the occurrence of the

combination of interactions in various MSCs of the system, which should be applied in the detection methodology.

Extension III:

There is a set of active processes $P_a = \{p_{1a\mathcal{M}_i}, \dots, p_{na\mathcal{M}_k}\}$ in a set of MSCs $\mathcal{M} = \{\mathcal{M}_i, \dots, \mathcal{M}_k\}$ of the system. The cases mentioned in the previous section are valid and should be examined based on the number of active processes in the set of MSCs.

Extension IV:

In this case, the number of active processes is more than two in some MSCs of the system. This case is studied in next sections.

5.3.2.2 Causes

Various reasons can cause the occurrence of implied scenario SLIS-II. The asynchronous concatenation of MSCs, having no timings on the MSCs to start or finish, timing issues for the interactions of active processes in the MSCs, and lack of conditions to start the interactions of active processes in the MSCs, are among the causes of this type of implied scenario.

The main issue lies in the behavior of the receivers of the messages that active processes send in these MSCs. If the receiver has some other interactions in the form of sending a message, it can cause an immediate problem in the system. However, if the receiver has no send interaction in these MSCs, it may or may not cause a problem, depending on the situation. If the new scenario exists in the MSCs of the system, then no implied scenario has occurred. Otherwise, the combination behavior will result in an implied scenario. These are discussed in detail in each of the cases Case I and Case II. In these cases, we have investigated the behavior of the receiver considering three different situations: the active processes send messages to each other, the

receivers are different processes, or the active processes send messages to the same receiver process.

5.3.3 SLIS-III: Non-local branching choices

This implied scenario is referred to as non-local branching choice in the literature [17, 93]. In this type, there is an hMSC that has two or more branches. There is a branch in the hMSC, when an MSC of the system accomplishes, and there is an option in the next MSCs of the system that should be followed. In each branch, some other MSCs of the system are determined to represent the system functionalities. Implied scenario **SLIS-III** occurs when a set of processes follow one branch and the other processes follow the other branch. The result is not in conformance with any of the MSCs of the system. The term "non-local" is added to the "branching choice" in the literature; since the branches are followed by various processes and the processes that decide which branch should be followed, are different in each branch. Therefore, the branching is not a local decision, and does not depend on the decision of one process. We have studied the case only one process can follow the branched as local branching choice in CLEB-III.

5.3.3.1 Specification

Consider the hMSC of the system $\mathcal{G} = (P, M, \boldsymbol{\mathcal{M}}, V, Ed, \mathsf{C}, F_0, F_f)$ that contains two branches $\mathcal{G}_1 = (M_1, \boldsymbol{\mathcal{M}}_1, V_1, Ed_1, \mathsf{C}_1, F_{0_1}, F_{f_1})$ and $\mathcal{G}_2 = (M_2, \boldsymbol{\mathcal{M}}_2, V_2, Ed_2, \mathsf{C}_2, F_{0_2}, F_{f_2})$.

The following conditions should be satisfied that SLIS-III occurs:

In $\mathcal{G}_1$, the first MSC is $\mathcal{M}_1 = F_{0_1}$ and process p is an active process in this MSC ($p_{a\mathcal{M}_1}$).

In the other branch, $\mathcal{G}_2$, the first MSC is $\mathcal{M}_2 = F_{0_2}$ and process q is an active process in this MSC ($q_{a\mathcal{M}_2}$).

And $p \neq q$.

SLIS-III occurs if process p follows $\mathcal{M}_1$ in $\mathcal{G}_1$ and process q follows $\mathcal{M}_2$ in $\mathcal{G}_2$, meaning that the hMSC in the execution time is neither following in $\mathcal{G}_1$ nor $\mathcal{G}_2$. Therefore, a new scenario is implied to the system. The existence of the above mentioned conditions represents the potential occurrence of SLIS-III. Although there might be a case that the combination of behavior of the two active process in the two SDs and other functionalities of the rest of processes (the ordered interactions) exist, there is still a chance of not executing the desired functionality of the system, because the scenarios will not be executed in the defined order.

The general case of SLIS-III is when we have more than two branches in the high level execution of the scenarios of the system.

The situations explained above are shown in Figure 22. In the left side of the figure, a sample hMSC is shown with a branch on MSCs M2 and M3. In each of these MSCs, there is an active process, namely p and Q (the first message is in Gold). Therefore, in the branch, the choice is given to different processes to decide which MSC to continue.

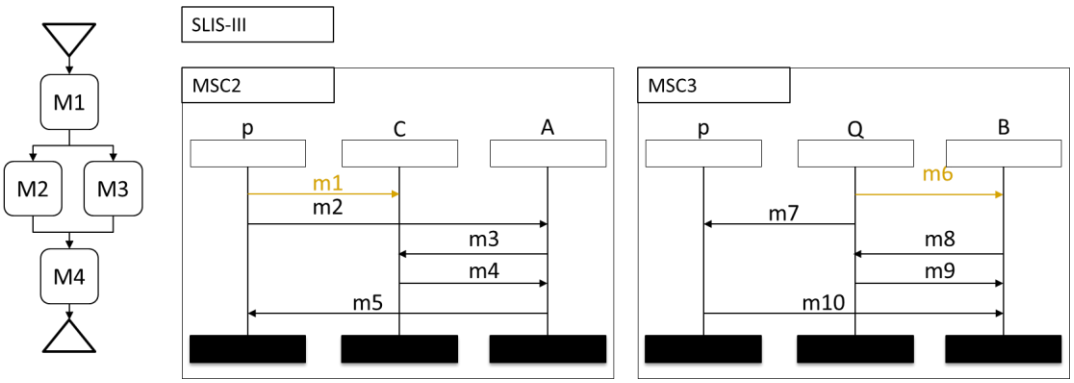

Figure 22. SLIS-III

5.3.3.2 Causes

In implied scenario **SLIS-III**, there is no control or global condition on the processes and the execution of the MSCs. Consider a case that the branch should be decided based on a timing

condition. If the timing is not global for all processes, or in an asynchronous communication, the timing is different for the active processes in each branch. Therefore, although the timing condition is applied on the branches, there is a chance of occurrence of the implied scenarios of SLIS-III. This condition can be other issues other than timings. In each case, the condition should be global for all processes, or all active processes in the first MSCs of each branch should be informed properly (globally), to prevent the implied scenarios of this type.

5.3.4 SLIS-IV: Asynchronous concatenation

Asynchronous concatenation of MSCs is referred to as different timings that processes can execute MSCs of the system. In other words, there is no blocking or waiting function specified for the processes that prevents the execution of other functions for each process. Therefore, each process may proceed in next MSCs in different times. In the literature, this case is defined as concatenation of the functionalities of each process for their following MSC in the hMSC, while the processes do not wait for the others to finish their actions in the current executing MSC [37, 112]. This may cause various problems in the system if specific conditions are satisfied. We refer to as these implied scenarios as SLIS-IV. Various examples for this implied scenario is explained in other chapters. Therefore, we will not present figures for section.

5.3.4.1 Specification

Let $\mathcal{M}_x, \mathcal{M}_y \in \boldsymbol{\mathcal{M}}$ be two MSCs from the set $\boldsymbol{\mathcal{M}}$ that contains all MSCs of the system, and $\mathcal{M}_x$ precedes $\mathcal{M}_y$ in the HMSC $\mathcal{G}$. Consider processes $p, q \in P_{\boldsymbol{\mathcal{M}}}$ where $P_{\boldsymbol{\mathcal{M}}}$ is the finite set of processes of $\boldsymbol{\mathcal{M}}$; and p and q have some actions in both $\mathcal{M}_x$ and $\mathcal{M}_y$. In asynchronous concatenation of MSCs, processes do not wait until all other processes accomplish their actions in one MSC. Therefore, while process p is still involved in $\mathcal{M}_x$, process q may proceed to perform its actions in $\mathcal{M}_y$. Implied scenario SLIS-IV can occur if one of the following cases are satisfied:

Case I: Let $\boldsymbol{\mathcal{M}}_{pa} \subseteq \boldsymbol{\mathcal{M}}$ as the set of MSCs in which a process $p \in P$ is an active process. An active process $p \in P$ can lead to SLIS-IV in all MSCs $\mathcal{M} \subseteq \boldsymbol{\mathcal{M}}_{pa}$ if there is no timing or control over the MSCs. The reason is that process p has the control of starting an action in a new MSC in $\boldsymbol{\mathcal{M}}_{pa}$.

Case II: Let process $p \in P$ in hMSC $\mathcal{G} = (P, M, \boldsymbol{\mathcal{M}}, V, Ed, \mathsf{C}, F_0, F_f)$ be an active process in $\boldsymbol{\mathcal{M}}_{pa} \subseteq \boldsymbol{\mathcal{M}}$, which has high level structure $\mathcal{G}_p = (M_p, \boldsymbol{\mathcal{M}}_p, V_p, Ed_p, \mathsf{C}_p, F_{0_p}, F_{f_p})$. Process p can lead to SLIS-IV if the following conditions hold:

- Process p is an active process in $\boldsymbol{\mathcal{M}}_{pa} \subseteq \boldsymbol{\mathcal{M}}$.
- And the $\mathcal{G}_p$ is a sub-graph of $\mathcal{G}$ and $\exists v \in V_p$ $such\ that$ $\mathsf{C}_p(v) \subseteq \boldsymbol{\mathcal{M}}_{pa}$.

Definition 4. Internal loop

A loop in $\mathcal{G}$ is an internal loop when it does not include the initial and termination vertices F_0 and F_f.

Case III: Consider an active or passive process p that has a loop in its high level structure $\mathcal{G}_p = (M_p, \boldsymbol{\mathcal{M}}_p, V_p, Ed_p, \mathsf{C}_p, F_{0_p}, F_{f_p})$ from F_{f_p} to F_{0_p}. If this loop is an internal loop in hMSC $\mathcal{G} = (P, M, \boldsymbol{\mathcal{M}}, V, Ed, \mathsf{C}, F_0, F_f)$, process p can cause SLIS-IV.

5.3.4.2 Causes

The difference in the high level structure $\mathcal{G}_p$ of an active process p and the structure of hMSC $\mathcal{G}$, which is defined as a global view for all processes, can lead to SLIS-IV. This difference can be having either various MSCs as the initial MSCs in $\mathcal{G}_p$ and $\mathcal{G}$, or having different termination MSCs. The former results in execution of the high level structure by process p, without following the scenarios that the whole system performs. In the latter situation, the active process p starts its initial MSC when the other processes start performing their scenarios. However, it does not follow the

same scenarios that other processes are executing in other iterations of the hMSC; because the termination nodes (final vertices) are different. Consequently, process p can start performing its high level structure $\mathcal{G}_p$ again, while the other processes are performing actions in the rest of MSCs in $\mathcal{G}$. In general, neither the initial nor the termination MSCs of $\mathcal{G}_p$ and $\mathcal{G}$ are the same.

When p performs actions in an internal loop, it may execute the loop more than other processes. The termination MSCs are different in $\mathcal{G}_p$ and $\mathcal{G}$. Therefore, p can start performing $\mathcal{G}_p$ again, while the other processes are continuing to perform their actions in other MSCs in $\mathcal{G}$.

5.4 Summary

In this chapter we represented one of the main contributions of this work: Catalogue of EB/IS. In this catalogue, we classified the main EB or IS that can happen in MAS and DSS based on our studies and various works specified in the literature. This catalogue identifies the issues into two main classes: Component level and System level. Each class has some sub-classes. In total, eight main categories are defined as various types of EB/IS that can happen in the system. This categorization is based on the specific conditions that can lead to an EB/IS. Therefore, the origins of the problem are analyzed and introduced in each class. The definitions and specific conditions that must be satisfied for each class are explained in this chapter. In summary, the eight classes are:

Component level emergent behavior (CLEB): Shared states, respond to different components, local branching choices, and race conditions.

System level implied scenario (SLIS): Shared interactions, behavior combination, non-local branching choices, and asynchronous concatenation.

Chapter Six: **Detection Methodology**

6.1 Introduction

In the previous chapter, we introduced the catalogue of emergent behaviors and implied scenarios that can happen in DSS and MAS. We have classified the EB and IS based on the origins of the problem and the level that they should be analyzed (component or system level). Also, we have mentioned our methodology using interaction graphs for the modeling of the behavior of the software components and the whole system and its advantages and the reasons behind this modeling. In this chapter, we will explain the general techniques for the detection of the specified EB/IS classes. The algorithms are devised for a set of the EB and IS in the catalogue. The details of the techniques used for the detection of EB/IS are explained in the following sections.

6.2 Problem definition

The detection of emergent behaviors or implied scenarios is not easy, since the number of states grow exponentially. On the other hand, the analysis of the states for the set of behaviors of each component and also the combination of the behaviors of all components requires analysis of all paths that exist in the system. This is interpreted as the language that can be generated by the alphabet which is modeled from the behavior of the components (referring to automata theory) and therefore all the language (paths) should be analyzed in order to check whether the defined properties are violated or some words exist in the language that is not consistent with the modeled system. Analyzing all of these states is computationally expensive. Consequently, various approaches such as bounded model checking or abstract methods are developed in order to restrict the number of states in the state space. The list of advantages and disadvantages of these approaches can be found in Chapter Two. Each of these approaches should be refined carefully in order to define the exact properties of the system under analysis. Moreover, in the existing

approached, when the EB/IS is detected, the explanation is language specific and there is no way to use it as a solution for other cases. Therefore, to overcome some of the problems with these approaches for the detection of EB/IS for large scale systems, new techniques should be used.

6.3 Methodology

The modeling approach used for the behavioral modeling of the components and the system can resolve some of the modeling problems for larger systems. In order to model and analyze the model of the system, we use the following strategies:

1. Restricting the conditions under investigation
2. Modeling the whole system once
3. Categorizing the problems
4. Modeling the component's view from the whole system

The first strategy is defined in the classification of common EB/IS in the system. In each class, the reasons and the conditions that can lead to an EB/IS are studied. In many of the cases, the functionality of an active process or sending a message in certain conditions can be the reason of emerging a new behavior. By specifying these conditions, restrictions can be applied to the states of each component or the model that should be analyzed. This technique can be achieved by the modeling that we have done in previous phases; since, in the interaction graphs, the information about the interactions of one component, the information about the states on each component, and the information about the type of messages that should be communicated in each state are preserved. Consequently, the specified conditions can be analyzed through the model of the components and the system.

The modeling strategy in our work requires modeling the system just one time, and then extracting the required information for the analysis. By modeling the whole system only one time,

we mean model the behavior of each component and the whole system at the same time, without requiring to model each component's behavior and then add their models all together as the system behavior. The interaction graphs gives us this flexibility to model all the required information and then extract the part of information needed for analysis of each part.

The third strategy we have chosen is classification of the problems. The amount of information that should be used for the analysis of component and system level is different. Also, in each level, we can do analysis on a specific part and if required use the other data to accomplish the task. This strategy helps to use small parts of the information which will help to prevent overloading problems. An example of this strategy is when we first analyze the shared states of a component, and if specific conditions are satisfied, then we analyze its interactions to detect the existing EB/IS.

The other strategy that we have used in our work is modeling the system behavior from the point of view of a single component. In other words, when considering the hMSC of the system, we can analyze how each component is going to execute its functionalities in various scenarios. One example of this case is used in the detection methodology of CLEB-III. This kind of modeling also helps in providing general solutions for various systems.

In the following, we will explain the general detection methodology and algorithms for component level emergent behaviors and we will explain the steps required for the detection of implied scenarios in some of the system level classes. It is worth mentioning that different functions used in the algorithms can have various implementations with different time and complexities. However, we do not go into detail for this part, since the main contribution of our work is the modeling and the classification of EB/IS.

6.3.1 Component level

The component level emergent behavior (CLEB) class is divided into four sub-classes as specified in the previous chapter: CLEB-I: Shared states, CLEB-II: Respond to different components, CLEB-III: Local branching choice, and CLEB-IV: Race conditions.

6.3.1.1 CLEB-I

6.3.1.1.1 Detection methodology

In order to find all the emergent behaviors of **CLEB-I** for each process p, we should find all the existing shared states between all scenarios of the system that process p participates in them. The next steps to check conditions against the above mentioned cases to find the emergent behaviors of CLEB-I. The steps to find whether a component has **CLEB-I** emergent behavior are as follows:

Algorithm I.1: $CLEB\_I\_Detection()$

Input: Sets of shared states for each component in all SDs of the system. $Set_i = \{set_{i1}, set_{i2}, \dots, set_{in}\}$ is the set of shared states of component P_i.

Output: List of components that have CLEB-I, determining the problematic states and SDs: $CLEB_{I_P} = \{CLEB_{I_{P_1}}, \dots\ CLEB_{I_{P_n}}\}$

1. For each of the components do the following
2. For each set set_{in} in Set_i
3. $CLEB\_I\_CaseI()$
4. $CLEB\_I\_CaseII()$
5. $CLEB\_I\_CaseIII()$

In the first algorithm (Algorithm I.1), all shared states Set_i among all the MSCs of the system for each component P_i are given as the input. We should mention that the shared states for one component can be more than one set (vector), since different states between various MSCs might be considered as shared states for each component. For each set of shared states for each

component, all the three cases are analyzed to detect which states in which scenarios of the system can make CLEB-I. These cases are examined in the following two algorithms.

Algorithm I.2: $CLEB\_I\_CaseI()$ and $CLEB\_I\_CaseII()$

1. For each SD in $shrdSD_{in}$ //SDs containing set_{in}
2. $MType = CheckMType(\theta_n)$ //Check the type of the message in the last shared state
3. If $MType$ is $Send$
4. Add (SD, θ_n) to $CLEB_{I_{P_i}}$
5. $MType = CheckMType(\theta_{n+1})$ //Check the message type of the immediate state after the shared states
6. If $MType$ is $Send$
7. Add (SD, θ_{n+1}) to $CLEB_{I_{P_i}}$

In Algorithms I.2, the message type of the last shared state (line 2) and the immediate state after the last shared states (line 5) for each set are examined. If this is of type Send (lines 3 and 6), i.e. the component is sending a message in these states, then the state and the associated MSC are added to the list of points that can cause an emergent behavior of CLEB-I.

Algorithm I.3: $CLEB\_I\_CaseIII()$

1. For each state θ_i in set of shared states Θ_p
2. For each SD in $shrdSD_{in}$ //SDs containing set_{in}
3. $MType = CheckMType(\theta_i)$ //Check the type of the message in that state
4. If $MType$ is $Send$
5. If $i \neq n$ //The state is not the last shared state
6. Add $MReceiver$ to $RList$ // add the receiver of the message to a list
7. If $|RList| > 1$
8. Add (SD, θ_i) to $CLEB_{I_{P_i}}$

In Algorithm I.3, case III is checked. For each state in the list of shared states, we check the message type associated to this state in all the related SDs (the MSCs that include this state) (lines 1-3). If the message is of type send, the we add the receiver of that message to a list (lines 4-6). If the number of the processes in the receivers list for each state is more than one, it means that in

two or more of the scenarios of the system, the component under analysis is sending a message (in one of its shared states) to more than one process, which can cause an emergent behavior CLEB-I.

The decision tree for the second and third cases are shown in Figure 23. In this figure, M stands for the message sent or received in each state. Based on the message type, the position of the state in the shared state vector, and the receiver processes of the messages, we can determine if the state will cause an emergent behavior of type CLEB-I.

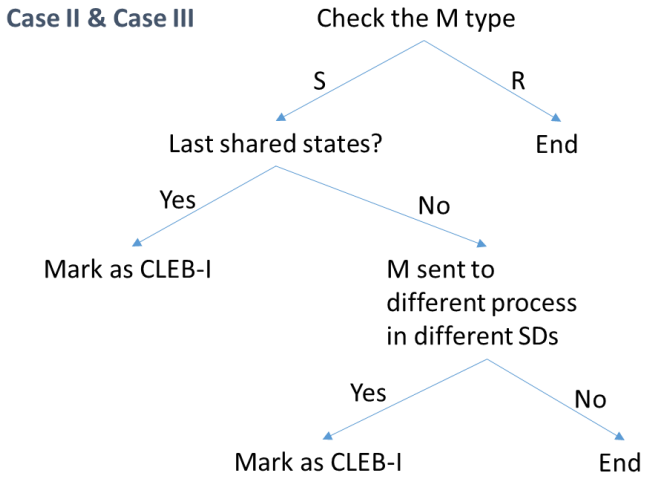

Figure 23. Decision tree to find case II and case III of CLEB-I

6.3.1.1.2 Case study

An order delivery system with scenarios in Figure 24 is chosen as the case study for CLEB-I [98]. In the first scenario (A), a client requests product A from the seller agent, and the seller sends the delivery order to the delivery department. In the second scenario, product B is requested by the client and then sent to the delivery department by the seller agent.

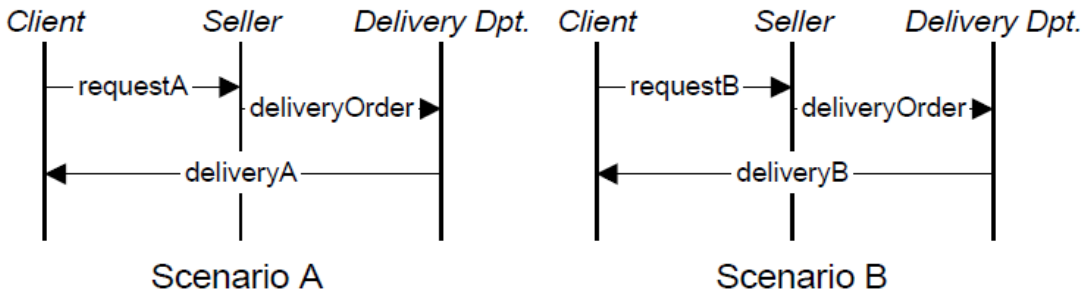

Figure 24. An order delivery system [98]

A possible emergent behavior is shown in Figure 25. In this scenario, the requested product (A) is different from the product that is ordered for delivery (B). This is caused because of the shared states of the delivery component. The delivery component has shared state (deliveryOrder) in the two scenarios, which is followed by either deliveryA or deliveryB states. Since the delivery component does not have a general view of the MSC that is executing in the system, it can execute either one of these states, which can result in an emergent behavior.

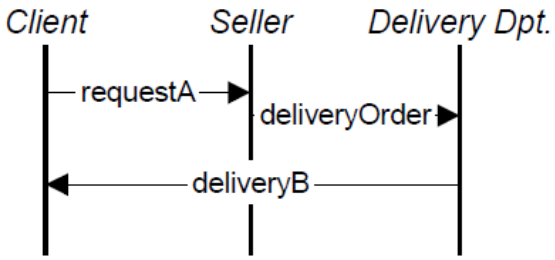

Figure 25. Implied scenario of Figure 24 [98]

Two other case studies, a web interface application and a toaster system from [98] are also categorized as shared states. Although the authors of [98] have different reasoning for the occurrence of the implied scenarios and emergent behaviors in these systems, we have defined them as having shared states that can cause an emergent behavior.

6.3.1.2 CLEB-II

6.3.1.2.1 Detection methodology

The emergent behavior CLEB-I and CLEB-II are similar in many cases. However, since we have categorized the emergent behaviors regarding various origins, they are separated to two different classes. Therefore, we can detect the origin of the problem, as well as developing solution repositories for each class separately. Since CLEB-I and CLEB-II have similar cases, we can take the advantage of the methodology used for the previous class of emergent behavior in CLEB-II. The extra steps required to detect CLEB-II are as follows.

Algorithm II.1: $CLEB\_II\_Detection()$

Input: Sets of shared states for each component in all SDs of the system. $Set_i = \{set_{i1}, set_{i2}, \dots, set_{in}\}$ is the set of shared states of component P_i.

Output: List of components that have CLEB-II, determining the problematic states and SDs: $CLEB_{II_P} = \{CLEB_{II_{P_1}}, \dots\ CLEB_{II_{P_n}}\}$

1. For each set of shared states in Set_i for each component do the following steps
   //checking condition I:
2. For each state θ_i
3. $MType = CheckMType(\theta_i)$
4. If $MType$ is $Receive$
5. For all SDs that have θ_i
6. $Sender = CheckSenders(\theta_i)$
7. If $|Sender| > 1$
8. Add (SD, θ_i) to $Marked_{I_{P_i}}$
9. If $Marked_{I_{P_i}} \neq \emptyset$
10. $CLEB_{II} - CaseI$
11. $CLEB_{II} - CaseII\&III$

In Algorithm II.1 we first check Condition I, specifying that if there is a shared state for process p in some MSCs, in which, p has same interactions of "receive" type with different

processes (lines 2-8). We then check cases I to III for the states that satisfy Condition I (lines 9-11).

Algorithm II.2: $CLEB_{II} - CaseI$

1. For each set of shared states
2. For each SD in $shrdSD_{in}$
3. $MType = CheckMType(\theta_n)$
4. If $MType$ is $Send$
5. Add (SD, θ_n) to $CLEB_{I_{i_{P_i}}}$
6. $MType = CheckMType(\theta_{n+1})$
7. If $MType$ is $Send$
8. Add (SD, θ_{n+1}) to $CLEB_{II_{P_i}}$

In Algorithm II.2, we check Case I is checked. We check the message type of the last shared state θ_n or its immediate next state in the MSCS. If this is of type Send, the scenario and the state are added to CLEB-II.

Algorithm II.3: $CLEB_{II} - CaseII\&III$

1. Find the number of shared states after the marked state $Marked_{I_{P_i}}$ in each set of shared states
2. If $Number > 1$
3. For each state θ_{i+1} // shared states after each of the states in $Marked_{I_{P_i}}$
4. $MType = CheckMType(\theta_{i+1})$
5. If $MType$ is $Send$
6. Add (SD, θ_j) to $Marked_{II_{P_i}}$
7. For each state θ_k in $Marked_{II_{P_i}}$
8. $PReceiver = CheckPRecName(\theta_k)$
9. Add $PReceiver$ to $PRecName$
10. If $|PRecName| > 1$
11. Add (SD, θ_k) to $CLEB_{II_{P_i}}$

In Algorithm II.3, we check Case II and Case III. For each state of type send after the states found for Condition I (lines 1-6), the names of the receiver processes are checked (lines 7-9). If the name of the receiver processes for each of these states is more than one, the state is added to the list of CLEB-II (lines 10,11).

For case II and III, no separate checking is required. Since case III is a general case for case II, therefore, case II is checked as well. Also, we should find the states that satisfy case II and III just for the detected state of condition I that has a lower rank. It means for example if we have two states s_i and s_j that satisfy condition I, and s_i comes before s_j in the shared states set ($s_i < s_j$), then, the only required check is to find the states that come after s_i and satisfy case II and III (for example s_k). The reason is that the detected state is either before or after s_j, but it is however after s_i. If it is before s_j, it cannot be detected in the checkings for states s_j. But, if it becomes after s_j, it is detected; because it is found in the list of states that come after s_i. Therefore, it is checked and found once. As a result, just one check for the all shared states is done, and it will not be a duplicate check.

6.3.1.2.1.1 Detection technique example

As we explained previously, for the functions in these algorithms, various methods and implementations can be used. Here, we specify one technique in which we use clustering. This technique is published in [153].

We extract the ${\Theta'}_p = \cup\, \Theta_p = \{\Theta_{p1}, \Theta_{p2}, \dots, \Theta_{pn}\}$ (set of all shared states of process) and then repeat all the following steps for each agent.

First clustering:

First, we cluster $\beta_p = \cup_{\mathcal{M}}\, \beta_{p\mathcal{M}}$ based on each member of ${\Theta'}_p$.

It will give us $n = |{\Theta'}_p|$ clusters, each one contains m vectors of β_p. Each state vector $\beta_{p\mathcal{M}}$ can exist in multiple clusters. Notation $\beta_{pi}\Theta_{pi}$ is used for the set of state vectors that have shared states vector Θ_{pi} in cluster i.

Then, we should examine the followings:

Find if $\theta_{ni} \in S$ or $\theta_{ni+1} \in S$ for the state vectors of $\beta_{pi}\Theta_{pi}$ in each cluster i.

Find if $\theta_{zi} \in S$ in the state vectors of $\beta_{pi}\Theta_{pi}$ in each cluster i.

We should omit the vectors that do not have any of the above conditions from further analysis. These examinations are for checking conditions indicated in Case I, Case II, and Case III (see section 5.2.2).

Second clustering:

Consider ${\omega'}_{pi}\Theta_{pi}$ for state vectors $\beta_{pi}\Theta_{pi}$ in cluster i that satisfy the above examinations and cluster them with edit distance.

In the second clustering if we have more than one group, CLEB-II occurs.

In order to find the roots of the EB, we extract the ${\varphi'}_{pi}\Theta_{pi}$ (shared state transition vectors) for state vectors in each cluster with various shared interactions. CLEB-II occurs because of various Θ_{pi} in cluster i.

The first clustering give us the information about the potential occurrence of CLEB-II, and the second one assures its occurrence.

6.3.1.2.1.2 Fixing the detected EB

Having different clusters means that p has various communications for a set of its states. We suggest solutions based on the results of our clustering:

1. If the number of groups for the vectors ${\omega'}_{pi}\Theta_{pi}$ in cluster i is more than two, it means there are more than two scenarios in which CLEB-II can occur. In this case, we suggest to change and modify the communications of this agent in the specified scenarios.
2. If the shared interactions for an agent equals to two, we can change the messages interacted or the agent saves the information of the senders of these messages.

The two solutions can be applied on any case study interchangeably. However, in case of priority of security or privacy issues, case should be taken for not revealing the information about senders of messages. Therefore, we can add acknowledgement messages or change the communicated messages to prevent the EB [153].

6.3.1.3 CLEB-III

6.3.1.3.1 Detection methodology

The following steps are done for each process in order to find if the process can have an emergent behavior of CLEB-III.

Algorithm III.1: $CLEB\_III\_Detection()$

1. For each process p do the following
2. $Branch = FindBranches(\mathcal{G}_p)$
3. $Shrd = SharedStates(Branch)$
4. If $Shrd \neq \emptyset$
5. Check for $CLEB\_I$
6. If the process is active ($p_{a\mathcal{M}}$) in at least one branch $\mathcal{M}$
7. If the process follows no condition in $\mathcal{M}$
8. Mark as $CLEB\_III$
9. If the process is active ($p_{a\mathcal{M}}$) in more than one branch
10. If the process has no conditions to follow at the branching state
11. Mark as $CLEB\_III$
12. If there is at least one other active process in one of the branches it is active
13. Yes: Check for $SLIS\_III$

Algorithm III.1 is applied on the high level structure of each process. For each process p, we check if the process has branches in its high level structure (line 2). If there is branches for the process, we check whether the process has some shared states in the branches (line 3). If there are shared states, we should check for CLEB-I emergent behavior (line 5). If the process is active in at least on the branches and it follows no conditions on that branch, then CLEB-III can occur (lines 6-8). If the number of branches that the process is an active process is more than one, and if the

process has no conditions to follow at the branching state, then we mark it as CLEB-III. If there are other active processes in one of the MSCs in the branches that the process p is also active, the case should be checked for implied scenario SLIS-III. Otherwise, it will not show CLEB-III (lines 9-13).

6.3.1.3.2 Case study

We reference this emergent behavior by an example case study (a boiler control system) from [83, 95] and the details which we previously published in [154]. The system is shown in the form of scenarios in Figure 26. The processes are a Control unit, a Sensor, a Database, and an Actuator (to control the pressure). The sensor information is stored in the Database. The Control unit uses the Database information to send commands to the Actuator. The four MSCs in this system are Initialize, Register, Analysis, and Terminate. There is also a hMSC that describes the organization of these MSCs.

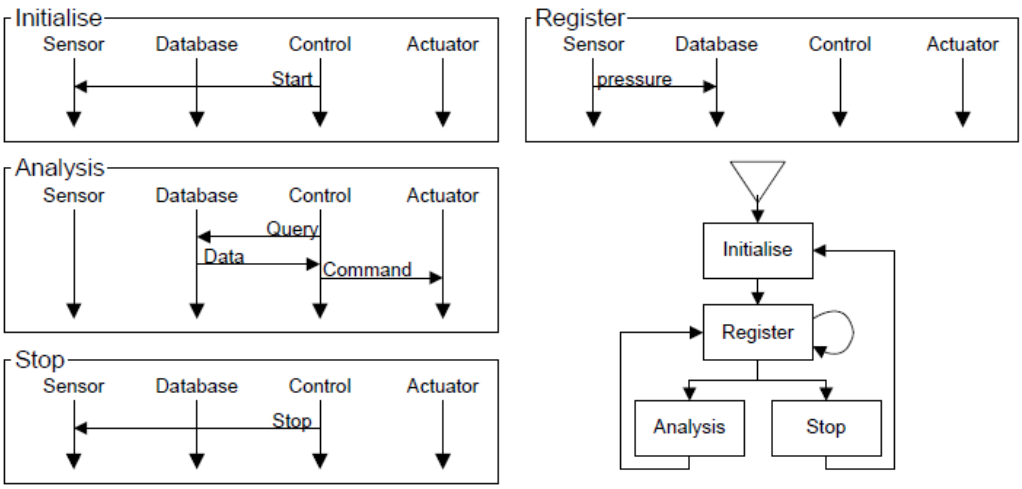

Figure 26. Four scenarios of boiler control system and hMSC taken from [95]

An emergent behavior MSC that can happen in this system is shown in Figure 27. As it is shown, the Control component sends the Query to Database. In this unwanted scenario, the

previous data registered to database by the sensor is used for computation. The sensor executes the initialize and termination MSCs and initializes again. However, the other processes analyze the data before the sensor is terminated, which is not acceptable.

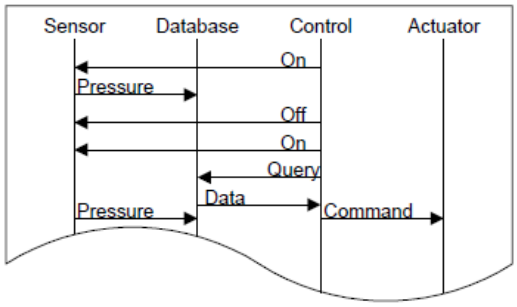

Figure 27. An emergent behavior of the system in Figure 26 [95]

If we consider the high level structure $\mathcal{G}_p$ of the components (Definition 7, Chapter Three), it can reveal the potential EBs in the system. For example, DB is only involved in Analyze and Register scenarios and there is no path between the other two scenarios for DB. As explained previously, the high level structure for each component represents how the scenarios are realized from the component's point of view. For designers, it is interpreted as the processes' states in the whole system and how the MSCs are executed for each component. Therefore, the paths that exist in the high level structure of each component which are not in accordance to the hMSC of the system are possible emergent behaviors.

We consider these paths here, since they are new paths from the local point of view of each component. One of these new paths is the loops in the internal MSCs of an hMSC. The component may execute the loop more than other components in the system. DB and sensor have such paths in this example [154]. The other new path is the ones for active processes. In this case, the path in the high level structure of the component has only the internal MSCs of the system, without the

initial and termination MSC. Here, the active process considers an MSC as its initial MSC which is not in conformance with the hMSC [154].

6.3.1.4 CLEB-IV

6.3.1.4.1 Detection methodology

Since we are certifying the receipt of messages in the execution time, we only consider the cases that a message is sent to a third process by an *active* process. The cases that are resulted by different timings, delays, message loss, and other related cases are not considered here; because we are examining the behaviors from system designs (MSCs) with regards to the visual orders specified for the scenarios. The timings regarding the asynchronous concatenations of MSCs is considered as an implied scenario SLIS-IV. Therefore, the only case that remains to make a problem can be the messages sent by active processes. This can be in various SDs that we have one active process in each SD that send a message to a third process p (implied scenario SLIS-II), or it can be in one SD that has more than one active process that send a message to a third process p.

Since we are comparing the new state vector against the state vector in the process's visual order, we should investigate the changes in the receive states of one process that are caused by two or more active processes. Then, compare this state vector with the defined (visual order) state vector of the process. In other words, we should mine what can happen, based on what is sent to a process by different active processes. The process of this analysis for one process is defined as follows.

Algorithm IV.1: $CLEB\_IV\_Detection()$

1. For each MSC $\mathcal{M}$ for process p do the following
2. $\quad ActiveP = FindActiveP(\mathcal{M})$
3. $\quad$ If $|ActiveP| > 1$
4. $\quad\quad$ For each $ActiveP_k$ in $ActiveP$
5. $\quad\quad\quad PReceiver = CheckPRecName(ActiveP_k)$

6. $PSharedRec = FindSharedP(PReceiver)$
7. For each $PSharedRec_k$ in $PSharedRec$
8. For each$ActiveP_i$ associated to $PSharedRec_k$
9. $newStateVector = FindNewStateVectorPRecName(ActiveP_i)$
10. If $newStateVector \in \beta_p$
11. For each $PSharedRec_k$ in $PSharedRec$
12. $newStateTransVector = FindNewStateTransPRecName(ActiveP)$
13. If $newStateTransVector \neg\in \varphi_p$
14. Mark as $CLEB\_IV$
15. Else
16. Mark as $CLEB\_IV$

In Algorithm IV.1, the number of active processes in each MSC $\mathcal{M}$ is counted (line 2). If the number of active processes is more than one, the other steps are done to find if emergent behavior CLEB-IV can occur. For each active process, the name of its receiver of its first state is extracted. Then we check if they have common shared receiver processes (lines 4-6). This gives a set of sets containing the names of the shared states associated with their senders. For each of the shared processes, we find its new state vector by changing the order of its receive message from the active processes one at a time (lines 7-9). If the new state vectors are in the set of state vectors β_p of the process, then the new state transition vectors should be checked against the state transition vectors φ_p of the process for its associated MSC (lines 10-12). If the state transition vectors are not the same, or the new state vectors are not in the set of state vectors β_p of the process, it has the potential to emerge a new behavior of type CLEB-IV (lines 13-16).

6.3.2 System level

The system level implied scenario (SLIS) class is divided into four sub-classes defined in the previous chapter: SLIS-I: Shared interactions, SLIS -II: Behavior combination, SLIS -III: Non-local branching choice, and SLIS -IV: Asynchronous concatenation.

6.3.2.1 SLIS-I

6.3.2.1.1 Detection methodology

The methodology to detect SLIS-I is the same as the methodology to detect its component level emergent behavior CLEB-I.

Lemma

If a set of processes have some shared interactions in a set of MSCs, each of the processes have its own shared states in this set of MSCs. The set of shared interactions A and the set of shared states B of one process p over a set of MSCs can be either one of the followings:

1. A is a subset of B
2. A is a superset of B
3. A and B have some joint elements
4. A and B are disjoint sets

The shared interactions A is investigated with the CLEB-I methodology for each component p, since A is always a subset of the component's shared states set B.

Proof.

If A is a superset of B, or they have joint elements, or they are disjoint sets, it requires that there are some elements in A that do not appear in B. This is interpreted as the existence of some frequent states that are not part of any of the sets in B. Since B contains all the states that appear in SDs with a number greater than the defined support i.e. all the shared states among all SDs that the process involves in, none of the above criteria are true, and A must be a subset of B.

More explanation for each case is as follows:

If A is a superset of B, it requires a situation that some states that are shared in different SDs are not defined in the shared states. However, all the combinations of all the consecutive states are considered as shared states as well. Consequently, A cannot be a superset of B.

If A and B have some joint elements, it requires the existence of some states that occur as shared states but are not considered in B. For the reasons explained above, this cannot be true.

If A and B are disjoint sets, it means that there were some shared states with support greater than one that were not considered as the shared states. However, in the detection process, we start with single states and then combine them to find the superset of each state that occur in more than one MSC. Therefore, the states of A should be considered as one of the sets of the component's shared states.

For the reasons explained above, A is always a subset of the component's shared states set B and is analyzed as the component's shared states in CLEB-I analysis. When A is part of the component's shared states (equal size or less than the size of the set of shared states), these states are investigated for the conditions for SLIS-I in CLEB-I, even more restricted conditions are investigated for them in CLEB-I. Therefore, if the conditions for occurring SLIS-I are satisfied, it would be detected with the methodology defined for checking CLEB-I. □

6.3.2.2 SLIS-II

6.3.2.2.1 Detection methodology

To study the potential occurrence of SLIS-II we should do the following steps. We first find the active processes in the scenarios of the system. For each active process, the receiver process of their first sending action is added to a list. For each process in the receivers list, we check the two cases, whether it sends a message in that MSC, or it is just receiving messages in the MSCs. If the process is sending a message after the marked state, it is the second case and the cases II-1, II-2, II-3 should be checked. Otherwise, the cases I-1, I-2, I-3 should be checked.

6.3.2.3 SLIS-III

6.3.2.3.1 Detection methodology

The steps required for the detection of SLIS-III is as follows. First, the branch/branches in the hMSC of the system should be extracted. For each branch, we identify the active processes in the following MSCs (successor MSCs of the branching MSC). The high level structure $\mathcal{G}_p$ for each of the active processes should be extracted. Then we check for three conditions: 1) if the high level structure shows two different graphs as the hMSC from the point of view of each active process, 2) whether in each branch we have more than one active process, 3) whether there is any condition or restriction on the behaviors of the active processes in the branches. If the high level structure shows different graphs as the hMSC from the point of view of each active process, it can cause SLIS-III, since the processes are able to follow different MSCs as their hMSC. Also, if we have more than one active process in a branch, it can lead to an implied scenario. The reason is that the active processes do not depend on the actions of other processes and they can start a new MSC. And the last condition checks the restrictions of the behavior of the active processes and on starting the branches. The cases that can prevent from having an implied scenario SLIS-III is when the high level structure of the active processes is same, or we have restrictions on the behavior of the active processes in the branching MSCs.

6.3.2.3.2 Case study

The case study of this section identifies six scenarios in a web based application calles MyPetStore [83] shown in Figure 28. It includes business components *UserControllerImpl* and *ShopControllerImpl* which are shared among various scenarios (representing resource sharring, which is a common characteristic of concurrent applications). The hMSC of the system is shown in Figure 29. This figure represents the order of the MSCs that should be executed for the shopping functionality in the system. There is a default scenario called *init* that leads to the *login* scenario. Form *login*, the system goes into *prepare shopping*, and then it has three possible transitions:

logout scenario, *authenticate* user, go back to *prepare shopping*, which allows the user to states a new shopping. From *authentication* scenario the system can only follow the *do shopping* scenario and from there to *prepare shopping*, starting a new cycle. In the right side of Figure 29 an implied scenario is represented that is an unexpected logout. In this scenario, the user logs into the system, then prepared for a shopping and the system authenticated the user. However, the final step is that the user leaves the system. This sequence is not specified in the original scenarios of the system. Instead of logging out of the system, the do shopping scenario should be executed.

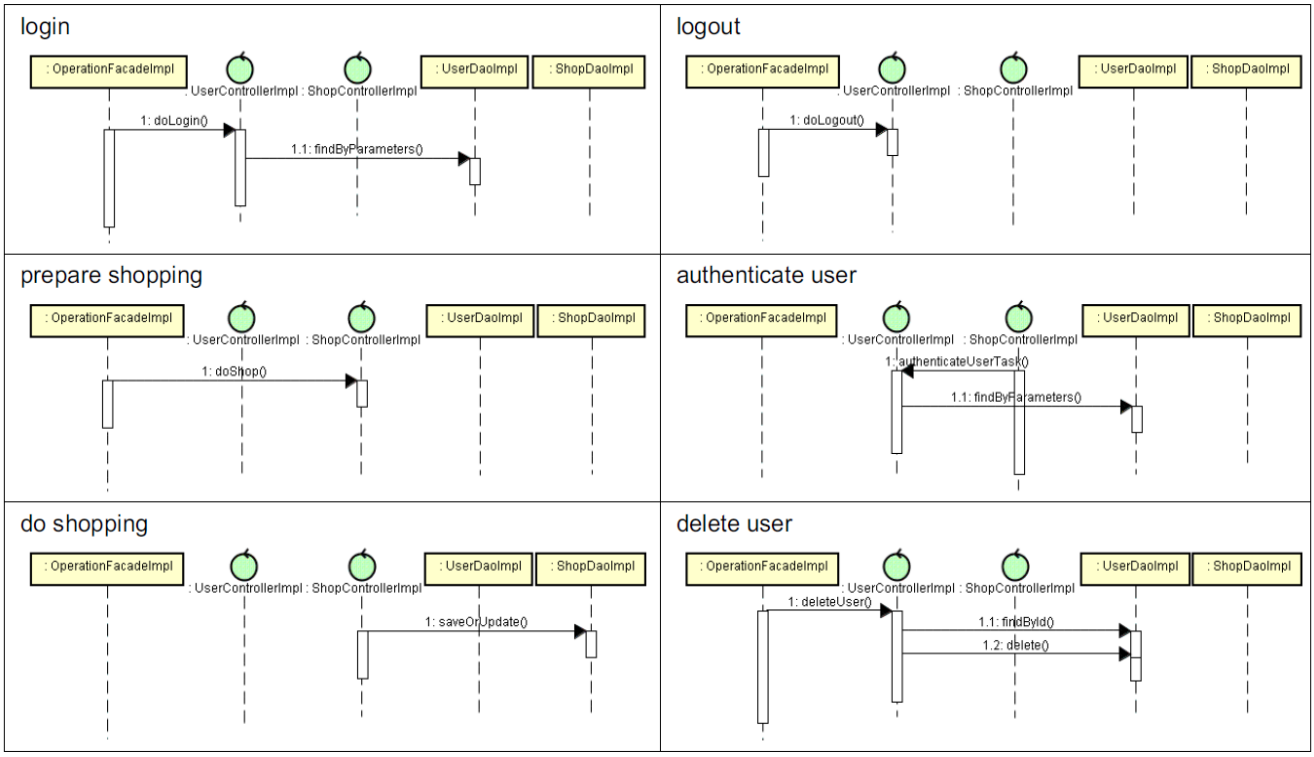

Figure 28. Six scenarios of MyPetStore web application [83]

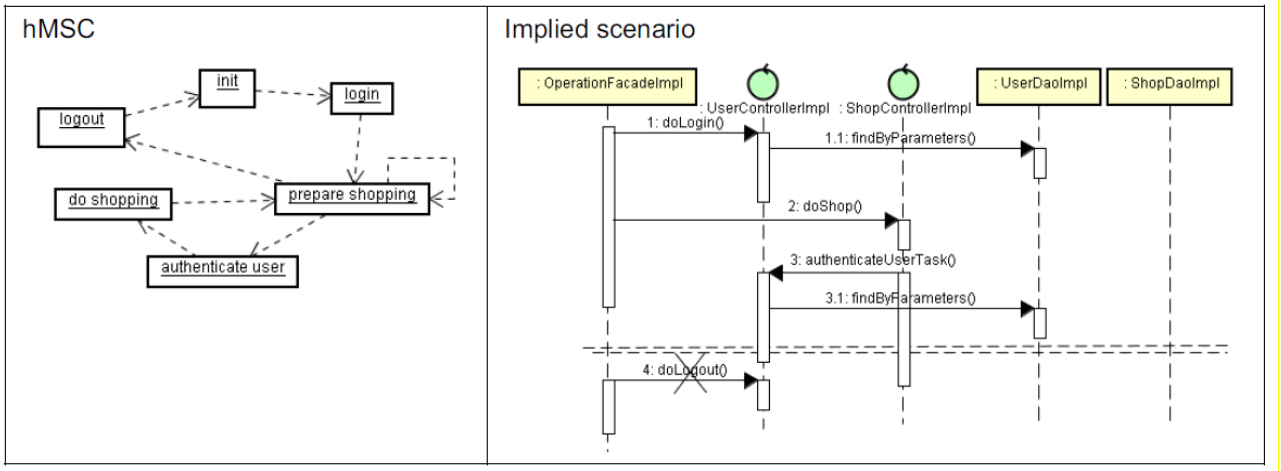

Figure 29. hMSC of MyPetStore application and the implied scenario [83]

We categorize this implied scenario as a non-local branching choice. From the modeling of the system as interaction graphs and the high level structure of the components, it is obvious that the *user* follows this sequence: *login*, *prepare shopping*, and *logout*. Also, the sequence for *ShopControllerImpl* component is *prepare shopping*, *authenticate*, and *do shopping*. As it is shown in the left side of Figure 29, there is a branch in the hMSC of the system after the *prepare shopping* scenario. One branch to the logout scenario and the other to the authenticate user scenarios. In each of these branches, we have one active process, namely *user* for the *logout* MSC and *ShopControllerImpl* for the *authentication* scenario. Since the decision of following the branches is given to two different components, this causes an implied scenario SLIS-III (nonlocal branching choice).

6.3.2.4 SLIS-IV

6.3.2.4.1 Detection methodology

The asynchronousness in a system can happen in one MSC or as the concatenation of the MSCs. For the former, the detection of this implied scenario requires a lot of process if we want to produce all the possible combinations of the order of messages in each MSC. One possible detection

methodology can be considering the order of the messages that if violated, it can cause major problems in the system. The next step is checking the conditions on the functionalities that guarantees these orders would be preserved. An example of these conditions is the awareness of the processes from the other processes' actions. As mentioned before, considering different combinations of the messages is another way to detect an implied scenario can happen or not. One way to reduce the complexity of producing and checking all possible combinations is grouping the messages that if their order if not preserved, it would not cause a problem in the system.

The other case is the concatenation of the MSCs. In this case, the detection methodology can be checked by restrictions required for the execution of functionalities of the processes, instead of executing parallel behaviors of the processes. Therefore, we can check if a wait function or a blocking send or received is defined for the parts of the system that requires synchrony between the functionalities of the processes.

There can be another implied scenario that is caused by the difference in the high level structure of a component (when it has loop) and the hMSC of the system. For this implied scenario, we should verify the high level structure $\mathcal{G}_p$ for each process against the hMSC of the system. If the sequence of MSCs in the $\mathcal{G}_p$ is different from the hMSC, meaning that there is a sequence that is not accepted in the hMSC, it can be an implied scenario. We refer to this detection methodology as Verifying MSC sequences against hMSC. This case is explained with the Boiler Control system (see section 6.3.1.3.2) in next section.

6.3.2.4.2 Case study

6.3.2.4.2.1 Case study I

The asynchronous concatenation of MSCs means that processes do not wait for other processes to start the execution of the next MSC. Also, it means that the processes do not follow a synchronous

timing to execute their actions even in one MSC. One simple example of this case is an ATM machine from the literature [98] which is shown in Figure 30. In this scenario, the user inserts a card, and after the pin is verified, user withdraws the requested amount.

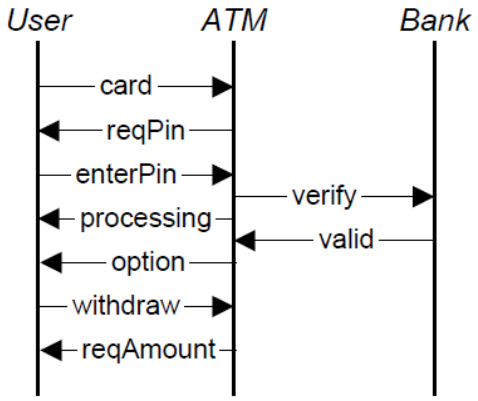

Figure 30. The withdrawal scenario of an ATM machine system [98]

An implied scenario regarding the asynchronousness of the functionalities of the processes is shown in Figure 30. In this scenario, the user is not notified about the process of validating its pin. Therefore, the *valid* message is sent to the ATM before the ATM sends the *processing* message to the user.

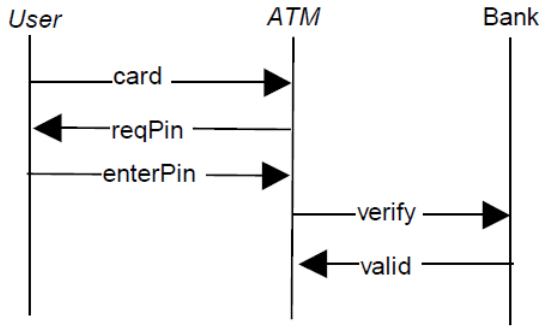

Figure 31. Implied scenario of Figure 30 [98]

Another example of this case is shown in Figure 32. The acceptable scenario is shown on the left and the related implied scenario is shown of the right side of the picture.

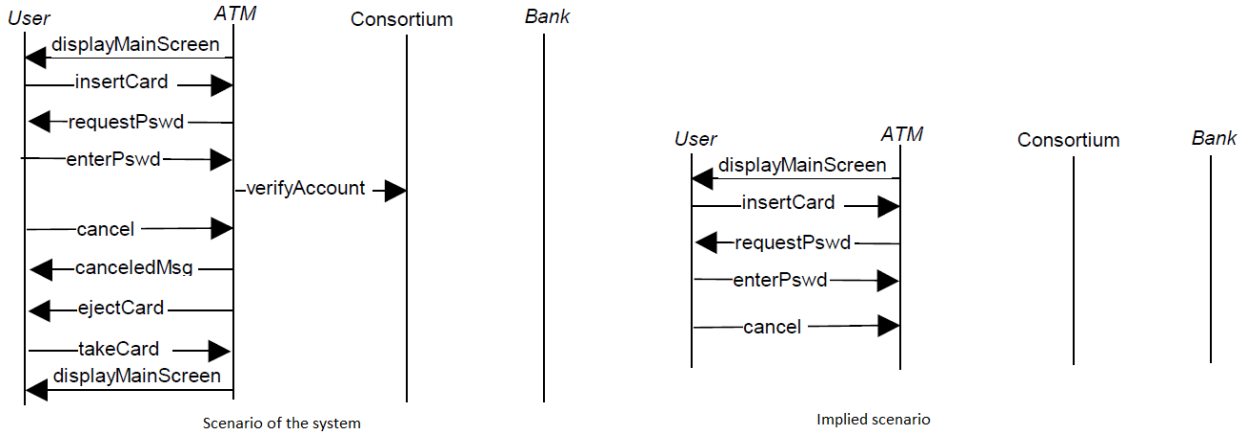

Figure 32. A scenario and an implied scenario of ATM machine [98]

6.3.2.4.2.2 Case study II

The MSCs and hMSC of the Boiler Control system is explained in section 6.3.1.3.2, so we avoid repeating it again.

We previously investigated two possible new paths that can cause an EB in the system in previous section, when there is an internal loop in the high level structure of a component or the high level structure of an active component does not contain the initial and termination MSCs of the hMSC of the system. The other condition that can cause an EB/IS is investigated here. We analyze the high level structure of the components by verifying their path of MSCs against the path of MSCs in the hMSC of the system. As explained in [154], the Control process's high level structure are: Initialize, Analysis, and Terminate, and execute Initialize again. However, there is not such a path in the hMSC of the system. One difference is that there is no link from the Analysis to Terminate in hMSC of the system. This difference can cause an IS in the system. The results of this part are previously published in [154].

6.4 Summary

In this chapter, we explained our methodology for analyzing the behavior of the system. For the EB Catalogue that we introduce in the previous chapter, we have developed detection methodologies. This detection method is defined for each class of the EB Catalogue separately. Therefore, the user can choose which type of EB/IS the system should be analyzed for. The methodology is a general methodology and only algorithms are provided. A class of these algorithms are implemented in the tool and by using various functions. In each algorithm, a key function is finding the shared states/interactions which can have different implementations. For each part, we have also included specifying the algorithm on a case study, mostly taken from other researches, to show step-by-step detection method. Also, for some classes, we have defined how to fix the detected problem in the system. This part (solution repository) is explained in each section of the catalogue. However, for some types of EB/IS, we have explained it in a separate section.

Chapter Seven: **Case studies**

7.1 Introduction

In this chapter, we present some case studies in which we show how to model the system, and how to detect some of the EB/IS based on the methodologies described in previous chapters. With these case studies, we also present how our modeling and detection techniques helps is suggesting solutions in each case.

7.2 Case studies

7.2.1 Fleet Management System

The first case study that we present in this chapter is a Fleet Management System, previously published in [153], containing *driver (user) agents*, *station*, *database*, *manager*, *GPS*, and *processor agents*. The system is responsible to track the *drivers* and plans for various routes, and estimates the departure or arrival time for the fleets as its minimum requirements. For working on the plans for predicting and scheduling tasks, the processor and the manager agents are interacting to each other. The former processes the times and performs the calculations, and the latter agent accomplishes the scheduling/rescheduling tasks. The four scenarios of the system are shown in Figure 33 to Figure 36. The first scenario, represents the communications of agents for schedule management, and the second one ($\mathcal{M}2$), demonstrates the prediction task.

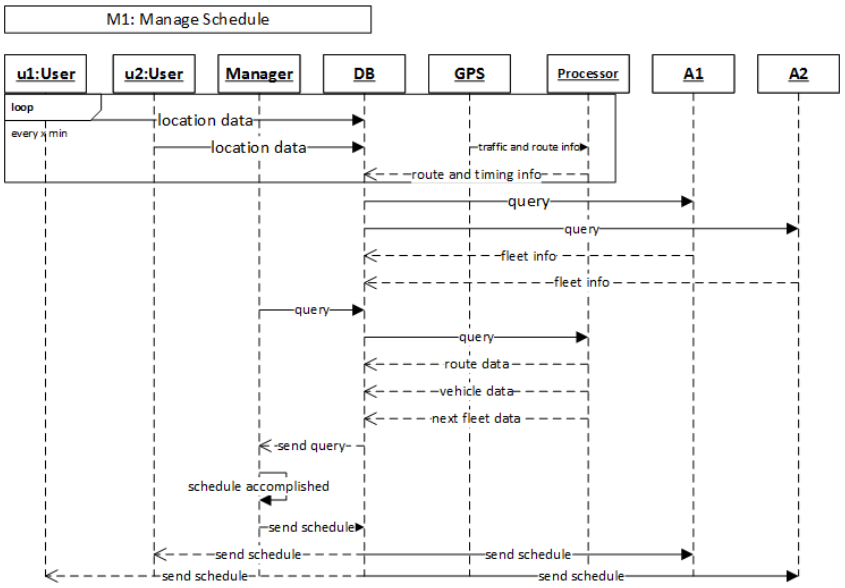

Figure 33. Schedule management [153]

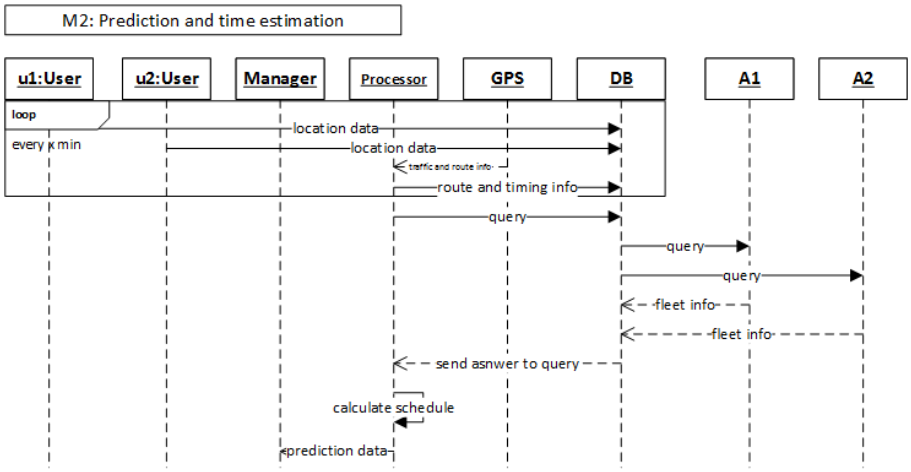

Figure 34. Prediction and time estimation [153]

The other scenarios are $\mathcal{M}3$ and $\mathcal{M}4$. In $\mathcal{M}3$, rescheduling tasks are done by sending requests to the manager agent. Also, the change in a schedule can be requested by the processor which is shown in $\mathcal{M}4$. Updating the information is done in a regular basis by the users, GPS, and the processor agent.

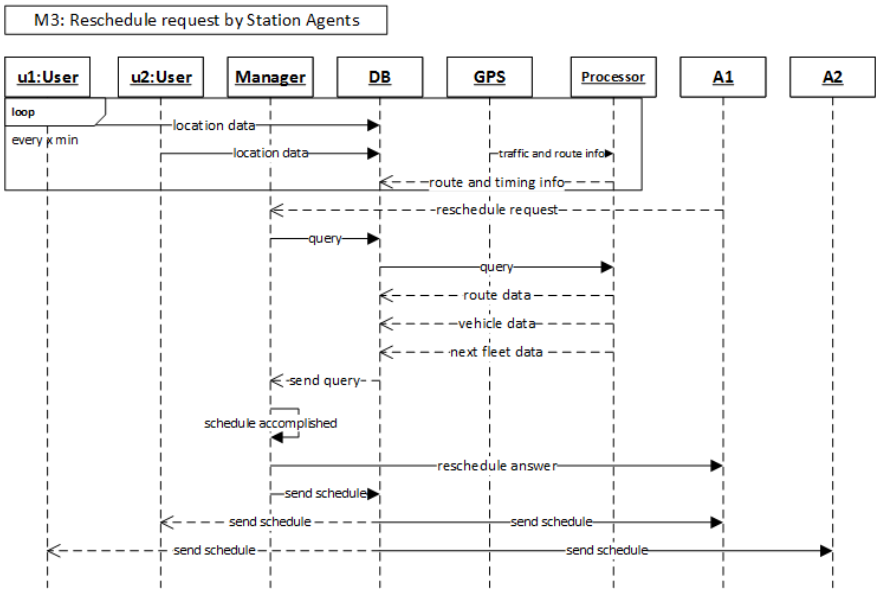

Figure 35. Reschedule request from station agents [153]

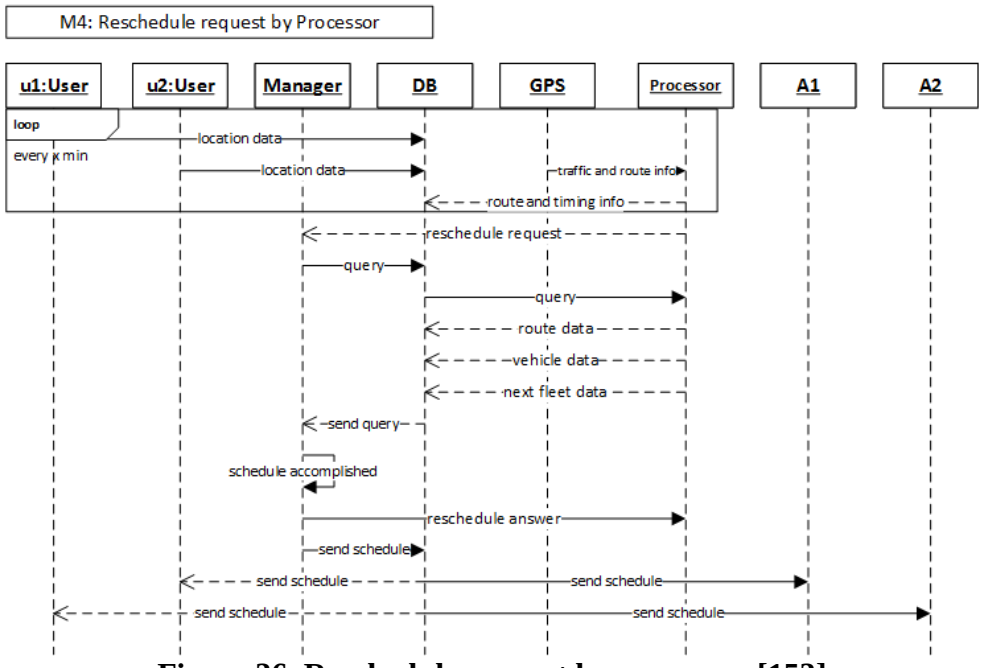

Figure 36. Reschedule request by processor [153]

The behavior of the *manager agent* is modeled as follows. The set of events for mng is $E'_{mng} = \cup_{\mathcal{M}} E_{mng\mathcal{M}} = (e_1, \ldots, e_n)$ which follows its global visual order ($\supset_{mng}$):

$\mu(e_1) = \text{mng! db(query)};$

$\mu(e_2) = \text{mng? db(send query)};$

$\mu(e_3) = \text{mng! mng(schedule accomplished)};$

$\mu(e_4) = \text{mng? mng(schedule accomplished)};$

$\mu(e_5) = \text{mng! db(send scheule)};$

$\mu(e_6) = \text{mng? proc(prediction data)};$

$\mu(e_7) = \{mng?\, q(reschedule\ request) | q = a1\ or\ q = proc\};$

$\mu(e_8) = \{\text{mng! q(reschedule answer)} | q = a1\ or\ q = proc\}.$

The important information that is modeled is the sender and receiver of each of the message labels (e_i) which is preserved in the Core and Node of the *interaction graph* for each of the scenarios. In our model, we use message labels and assign separate message labels to each content of the messages. We use events as the message labels here and we have

$E_{mng\mathcal{M}1} = (e_1, e_2, e_3, e_4, e_5);\ E_{mng\mathcal{M}2} = (e_6)\ ;\ E_{mng\mathcal{M}3} = E_{mng\mathcal{M}4} =$

$(e_7, e_1, e_2, e_3, e_4, e_8, e_5).$

The set of state vectors and shares states for the manager agent are:

$\beta_{mng\mathcal{M}1} = \{e_1, e_2, e_3, e_4, e_5\};$

$\beta_{mng\mathcal{M}2} = \{e_6\};$

$\beta_{mng\mathcal{M}3} = \{e_7, e_1, e_2, e_3, e_4, e_8, e_5\};$

$\beta_{mng\mathcal{M}4} = \{e_7, e_1, e_2, e_3, e_4, e_8, e_5\};$

$\Theta_{mng1} = (\theta_1, \dots, \theta_n) = (e_1, e_2, e_3, e_4);$

$\Theta_{mng2} = (a_1, \dots, a_n) = (e_7, e_1, e_2, e_3, e_4, e_8, e_5).$

As we have explained in the previous chapter, one possible detection methodology is the usage of clustering. Based on the defined methodology for CLEB-II, a two phase clustering is

done. Regarding the steps defined previously, the second step is to cluster the state vectors. The results is two clusters.

$$c1 = \beta_{mng1}\Theta_{mng1} = \{\beta_{mng\mathcal{M}1}, \beta_{mng\mathcal{M}3}, \beta_{mng\mathcal{M}4}\};$$

$$c2 = \beta_{mng2}\Theta_{mng2} = \{\beta_{mng\mathcal{M}3}, \beta_{mng\mathcal{M}4}\}.$$

The state vectors in each cluster are analyzed for two conditions explained previously. The state vectors in *c1* are $\{e_5, e_8\} \in S$. As a result, they should be analyzed for the second step. Also, for *c2* there are some events of type "send". Therefore, the second condition is satisfied and they should be analyzed as well. The shared interaction vectors for state vectors in cluster *c1* are:

$$\omega'_{mng\mathcal{M}1}\Theta_{mng1} = \omega'_{mng\mathcal{M}3}\Theta_{mng1} = \omega'_{mng\mathcal{M}4}\Theta_{mng1} =$$

$((mng, db), (db, mng), (mng, mng), (mng, mng))$;

And for cluster *c2* are:

$$\omega'_{mng\mathcal{M}3}\Theta_{mng2} = ((a1, mng), (mng, db), (db, mng),$$

$(mng, mng), (mng, mng), (mng, a1), (mng, db))$;

$$\omega'_{mng\mathcal{M}4}\Theta_{mng2} = ((proc, mng), (mng, db), (db, mng),$$

$(mng, mng), (mng, mng), (mng, proc), (mng, db))$;

The second clustering is done with Edit distance. This metric for the first cluster is zero, and therefore they are all classified into one class. This metric for the shared interaction vectors in second cluster is two and therefore we will have two groups for this category. As a result, we see that the interactions of mng in the third and fourth MSC are different, since it has some shared states. Accordingly, an emergent behavior can occur because of the reasons explained in previous chapters. To prevent this problem, it should differentiate for its states in Θ_{mng2}.

To explain more, the shared state transition vectors of the states vectors of the second cluster are:

$$\varphi'_{mng\mathcal{M}3}\Theta_{mng2} = \{a1, mng, db, mng, mng, mng, mng\};$$

$$\varphi'_{mng\mathcal{M}4}\Theta_{mng2} = \{proc, mng, db, mng, mng, mng, mng\}.$$

It is obvious that the senders are different for the first shared states and this reveals the roots of the occurrence of CLEB-II.

We suggest two solutions for this system. First, the mng can save the information of its sender/receiver in the detected states. Second, change these interactions by adding an acknowledgement or change the messages communicated to differentiate the states so that they will not be considered as shared ones any more.

7.2.2 Greenhouse System

This case is a Greenhouse Multiagent System which consists of three different types of agents: *Temperature balancing agents* (A_t), *Water control Agents* (A_w), and *Mineral control Agents* (A_m). The agents are responsible to control the environment of the greenhouse. They receive the environmental information of the greenhouse from sensors. Then, connect to data and knowledge bases (KB). Based on the data they receive; they analyze the information in order to be able to perform the best task. The agents communication to each other to manage the greenhouse environment in the best condition. They are responsible to manage the resources such as water and minerals, keep the temperature balanced for the greenhouse, save resources by interacting to other agents, and monitor and save changes based on various decisions in the KB. MaSE methodology (Multiagent Systems Engineering) which is developed by Deloach and Wood is used to design this system [155]. MaSE is meant to have a complete lifecycle for analysing, designing and developing heterogeneous Multiagent systems [155, 156]. It has two main phases: Analysis and Design.

Similar to object oriented paradigm, agents communicate through conversations to accomplish the goal of the system in a top down approach [155]. The system goals, as well as the agents and their tasks and conversations are modeled in seven steps in MaSE. The deployment step also helps the physical placement of agents and its implementation. This methodology is independent from agent architecture or programming languages and helps to track changes in all phases [155]. The steps of MaSE are shown in the left part of Figure 37 and are taken from [156]. The right section of this figure represents the application of our methodology.

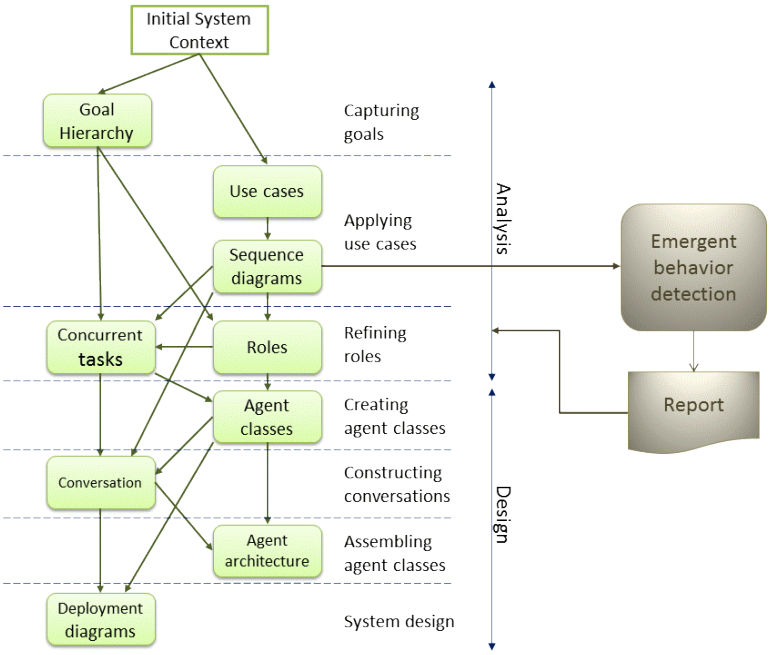

Figure 37: Emergent behavior detection in MaSE

We can use MaSE SDs or transform them to MSCs using the technique explained in [106]. In this technique, the roles of each agent are extracted from the agent class diagram. Then, the messages for each role are extracted.

Two scenarios of the system are shown in Figure 38 and Figure 39. The first one shows mineral balancing with water misting method. The second one demonstrates how temperature should be balanced either by mist method or use a thermostat.

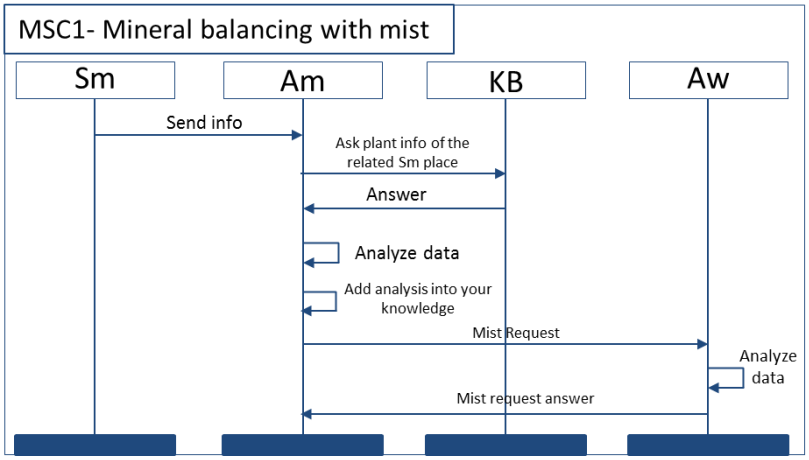

Figure 38. Balance minerals with misting [150]

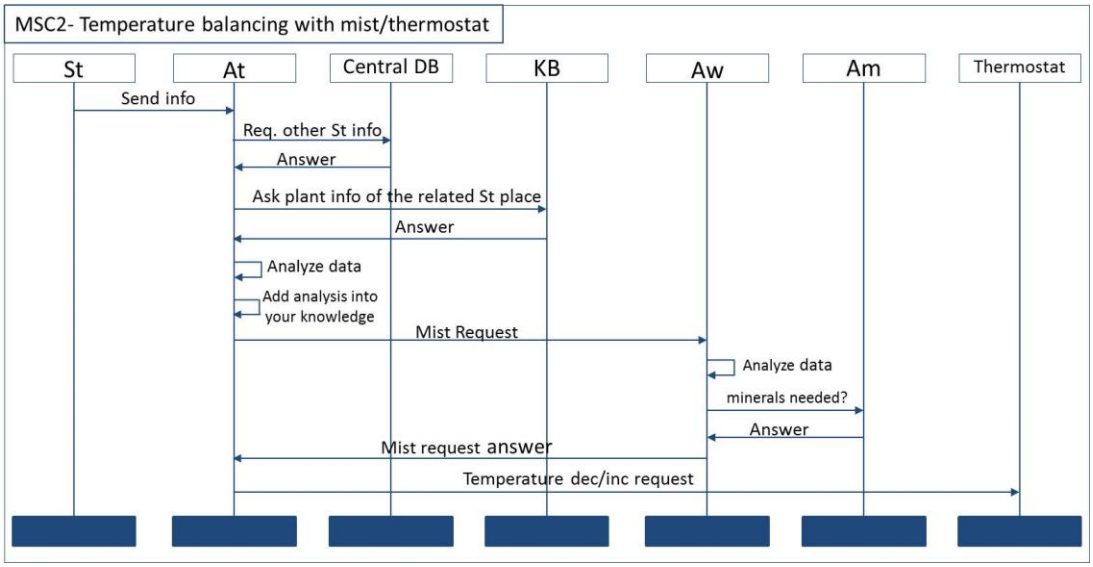

Figure 39. Balance temperature [150]

We can model the agents' behavior as interaction graphs or as send or receive matrices. If matrices are used (for smaller systems), the followings are the required definitions:

We model the interaction of processes in each MSC_i into its corresponding interaction matrix IM_i. An interaction matrix is a square matrix of size $n = |P|$ equal to the total number of processes in the system. Two types of interaction matrices are defined that are used for various communication channels: *Send Interaction Matrix* and *Receive Interaction Matrix*.

Definition 1. Send interaction matrix in an MSC SIM

A ***Send Interaction Matrix*** SIM over P is a matrix that represents all communications in an MSC $\mathcal{M}$ that are of type "Sending". The entries of SIM are $\{sim_1, sim_2, \dots\}$ where $sim_z \in S$ and row_p represents the p^{th} row in SIM and $row_p = \delta_{p\mathcal{M}}$.

Definition 2. Receive interaction matrix in an MSC RIM

A ***Receive Interaction Matrix*** RIM over P is a matrix that represents all communications in an MSC $\mathcal{M}$ that are of type "Receiving". The entries of RIM are $\{rim_1, rim_2, \dots\}$ $where\ rim_z \in R$ and row_p represents the p^{th} row in RIM and $row_p = \gamma_{p\mathcal{M}}$. In FIFO communications, with synchronous communications, and in cases that no message is lost (ideal case), we have $RIM_i = (SIM_i)^T$ for their corresponding MSC $\mathcal{M}_i$, i.e. the two matrices SIM_i and RIM_i are transpose of each other. However, in other communication channels, SIM_i and RIM_i for their corresponding MSC $\mathcal{M}_i$, might be different based on the definitions of that channel, and $RIM_i \neq (SIM_i)^T$.

For this case study, we use matrices. The vectors used for the analysis are the same for both modeling approaches. The information for agent A_w is followed. The information of the 5th rows and columns of the matrices are related to agent A_w. Vector $states_{5k}$ (states vectors of A_w) are:

$\beta_{A_w\mathcal{M}_1} = \{st_1, st_2, st_3, st_4\};$

$\beta_{A_w\mathcal{M}_2} = \{st_1, st_2, st_3, st_4, st_5, st_6\}.$

And state transition vectors are:

$\varphi_{Aw\mathcal{M}1} = \{A_m, A_w, A_w, A_w\};$

$\varphi_{Aw\mathcal{M}2} = \{A_t, A_w, A_w, A_w\ A_m, A_w\}.$

The first three states are shared states for A_w. The senders of these states in the first scenario are {A_m, A_w, A_w} and in the second scenario are {A_t, A_w, A_w}. This difference can cause a problem which is explained in previous chapters.

We can add arguments to the concurrent task and conversation steps of this agent in MaSE to store the information about sender/receiver of the detected states to prevent this issue. We can also revise the designs and check the designs again for an EB/Is. As it is shown in Figure 40 (the concurrent tasks of "Receive request" of the water control agent) argument *S* associated to Sender of a request in its states is stored.

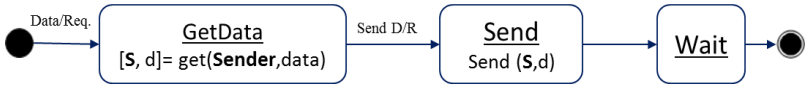

Figure 40. Aw concurrent task diagram [150]

The responder side of the conversation for temperature balancing is shown in Figure 41 (communication between A_t and A_w).

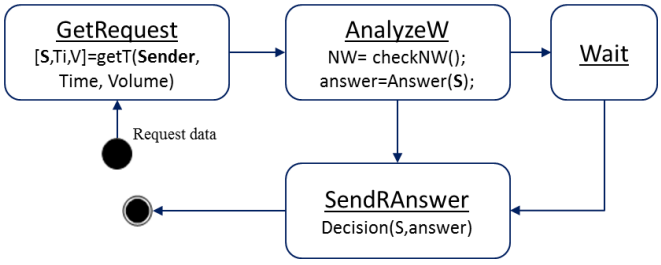

Figure 41. Conversation diagram of At and Aw [150]

To sum up, the process of our work and how it is applied on MaSE is shown in Figure 42.

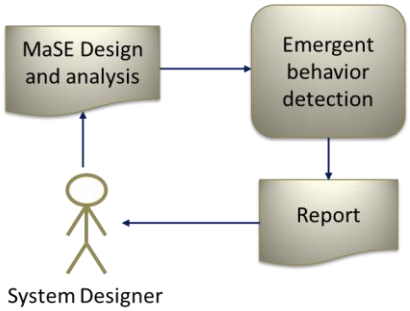

Figure 42: Process of EB detection and fixing in MaSE

If we want to use other approaches to detect the shared states, we can find identical states by techniques in [4, 151] or [157]. In this approach, FSM is used which is shown in Figure 43 and Figure 44. Messages m1-m5 are the abbreviation we used here for simplicity. As it is shown, the senders of the messages are written above them. Storing the senders' information is not shown in FSM and we show this information to clarify the importance of keeping all the information of the interactions among software agents in Mas.

Figure 43: Aw FSM in first scenario [158]

Figure 44: Aw FSM in second scenario [158]

The results of this case study are previously reported in [150, 158] and are taken from these papers.

7.2.3 Online Auction System

In this section, an online auction system is studied. We show that by this modeling and detection technique, the agents that have no emergent behavior can be omitted from further analysis. We chose an online auction system since in this system there are many agents of different types that

are interacting to each other and many of them are autonomous agents. This case is previously published in [147]. Considering it as a social network of business agents, individual sellers and buyers participate to provide various items and bid on the items. The agent based modeling models the auction system where each agent works as a buyer or seller. Learning and behavioral change can also occur [159]. We suppose that rational agents are working together with specific objectives and therefore they should follow rules defined by the auction hosting authority [160]. New behavior can be seen in the behavior of buyers and sellers agents based on their previous experiences [161].

Analysis of e-commerce systems is important for trust and security checking and fraud detection.

We only consider six agents here and other agents such as the security or trust manager agents are not considered to make the case study simple:

Controller or market place agent (C1): Defining the auction types and protocols for each one and defining the rules for each auction type is the responsibility of this agent.

Auctioneer (C2): This agent performs the auctions and declares protocols, seller and buyers, items in each auction, timing and pricing, bids, and other actions required in an auction.

Registrar (C3): The communication of seller and buyer agents with the auctioneer is through this agent. Actions such as introduction, registering new users, logging the functions, and searching is among the responsibilities of this agent.

Seller (C4): This agent sells items in auctions. It should register in each auction type through the registrar, then is able to announce its items and the prices.

Buyer (C5): The buyer agent searches for its wanted items, bids on various items, and buys them. It should register first and log into the system.

Credit Associate (C6): Securing the financial transactions by sending the seller and buyer information and the trade information to the financial associates.

Some of the scenarios for this system are shown in Figure 45-Figure 50. The scenarios are shown with message sequence charts.

Scenario 1: Registering a new user.

The auctioneer asks the auction information and the registrar shows the information on the web site. Next, seller or buyer send their information and register as a new user. This scenario is shown in Figure 45.

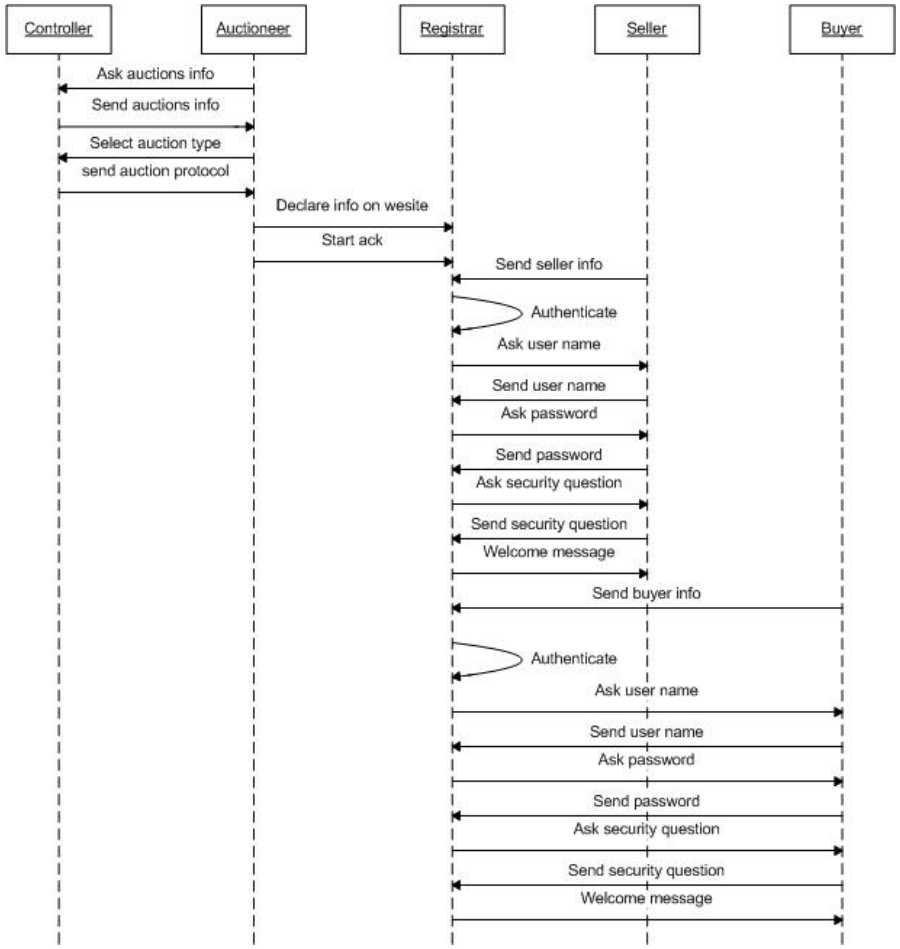

Figure 45. User registration [147]

Scenario 2: Users sign in.

The registered users participate in this scenario. They sign into the system and participate in an auction. This scenario is shown in Figure 46.

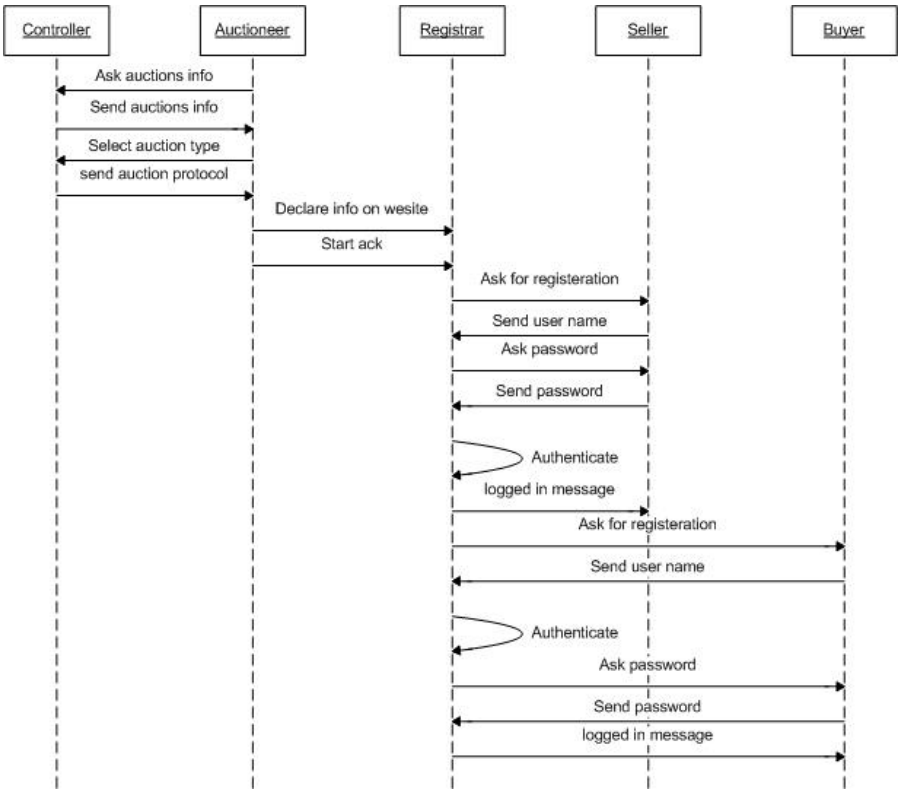

Figure 46. Users sign in [147]

Scenario 3: Registering users to participate into a certain auction.

The users signed into the system register for a certain type of auction. The registrar agent introduces these users to the auctioneer. Bidding, declaring the winner, and sending the information of the sold items and the trades to the credit associate agent are shown in this scenario. The third scenario is shown in Figure 47.

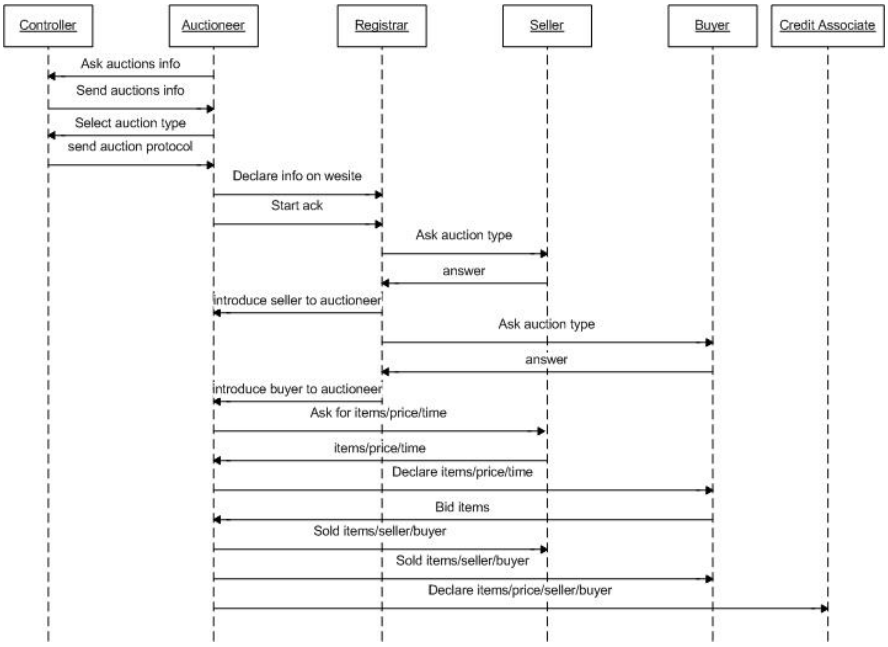

Figure 47. Registering to auctions [147]

Scenario 4: Searching for items.

This scenario is shown in Figure 48. In this scenario the buyer searches for an item. The search rules are also defined here.

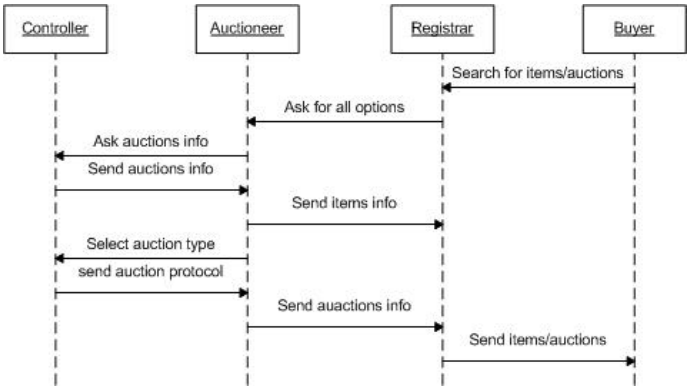

Figure 48. Searching for items [147]

Scenario 5: Searching for type of auction.

It is shown in Figure 49. The seller seller searches for various types of auctions and receives related information and rules of each auction. The rules and protocols of a chosen auction is declared.

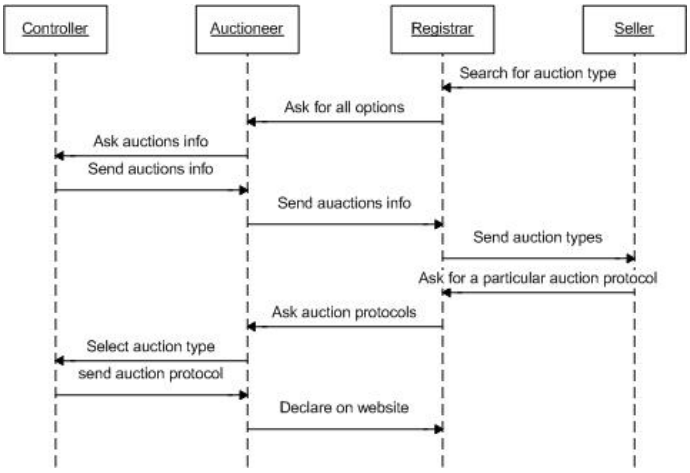

Figure 49. Searching for auction type [147]

Scenario 6: Registering items.

The seller agents declare the items and all the required information such as the price and the auction type to the registrar. The registrar agent is responsible to put all of the received information on the website. This scenario is shown in Figure 50.

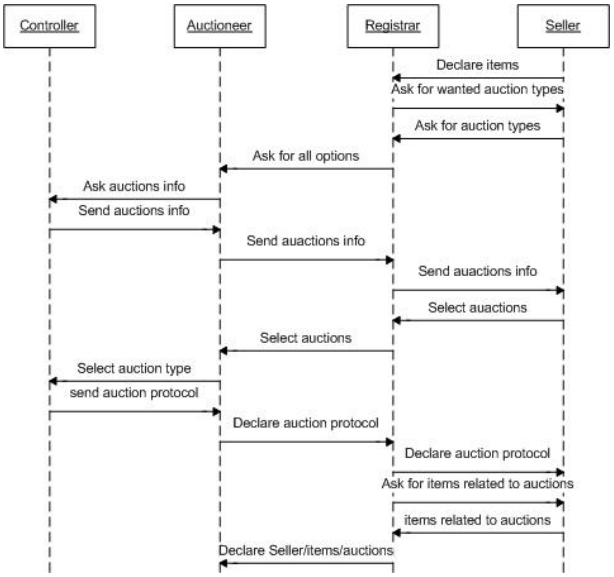

Figure 50. Registering items [147]

For the detection of agents that have no emergent behavior, we refer to the pre-processing phase in 0, the synchronous communication style. By applying the algorithm on the extracted vectors, we will find that the first and the last (sixth) agent have similar message labels in all of their MSCs (are in set $S_{imilarM}$).

For each of these components, we should find their send vectors vector $\delta_{p\mathcal{M}_j}$ in all the MSCs, and then compare these vectors together. If the send vectors of a component are similar in all MSCs, it is added to set $S_{imilarS.}$. For the components that pass this phase, the receive vectors $\gamma_{p\mathcal{M}_i}$ are checked against each other in all MSCs. If these vectors are the same, then component is reported as an agent with no EB.

The first agent, C_1, passes all the steps mentioned above in all the MSCs. Therefore, it is reported as a component that has the same behavior in all MSCs of the system and cannot show an EB. This component has the same behavior in all scenarios of the system, and that is the reason of having no EB.

The other component is the credit associate agent. This agent receives messages in all MSCs, without a send event on its functionality. For this agent, all the above mentioned steps are the same, and consequently it cannot emerge a new behavior in the system.

We should mention that this technique is also useful for the detection of neutral agents in a network of software agents; because the complex social network of the existing relationship between the agents taking part in a system such as in an online auction make the modeling and analysis of such a network complicated. Therefore, by detecting the neutral agents, we can model the rest of the system for other analysis. It can help in reducing the complexity related to analyzing such a network and analyze the relationship between different types of the agents before analyzing the whole network. The case study is part of our work which is published in [147].

7.2.4 Traffic Control System

In this section, we present a Traffic Light Control System which we published in [C1]. This system is a Multiagent system and is simplified to six agents working on two high ways. Through this case study, we will explain the problems that can arise because of asynchronous concatenation of MSCs and race conditions. It is worth mentioning that one of the reasons of having a race condition is the asynchronous communication style between the processes.

Figure 51 illustrates the traffic situation. It contains two highways HW-1 and HW-2 with two ramps that are exit of HW-2 and entrances to HW-1. The vehicles can do in the direction that is depicted with dashed arrows. The inflow and outflow of the ramps to/from HW-1 and HW-2 are controlled by ramp metering installations (RMI) RMI1 and RMI2. RMI2 is responsible to detect the congestion in the downstream of the second ramp (starts to build up on HW-1) and reduce the inflow. Likewise, when the congestion starts on the exit ramp of HW-2, RMI1 and RMI2 should act to reduce congestion. To increase the performance, the agents RMI1 and RMI2 should be aware

of future congestion. Therefore, the system is designed as a Multiagent System (MAS), where the agents can interact to each other. Therefore, they can take actions in advance and solve the congestion timelier.

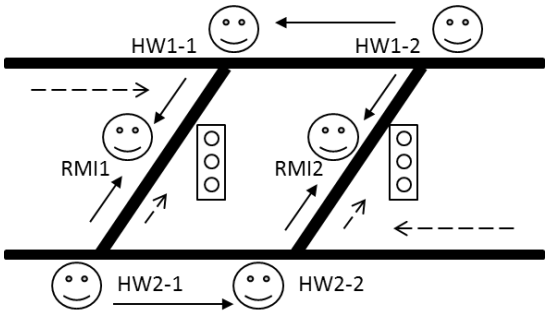

Figure 51: Roads control by MAS congestion control [C1]

There are two highway agents on each highway and one RMI agent for each ramp. The highway agents are able to develop an image of the current traffic situation, which is based on their estimation and the messages they receive from downstream agents. The possible communications among agents is shown with arrows in Figure 51. Highway agents can send these messages: “no problem”, “help-urgent high”, and “help-urgent low”. The actions for the RMI agents are: “make the traffic light green” or “make the traffic light red” which is based on the messages they receive from highway agents.

In the following, two MSCs are shown as possible scenarios of the system. These MSCs show situations that agents of HW1-1 and HW2-1 request urgent help from RMI1 agent in different orders. In scenario MSC M1, the message from HW2-1 is sent sooner. As a result, RMI1 agent makes the traffic light green. This is shown in Figure 52.

In MSC M2, Figure 53, HW1-1 agent’s request is received sooner and the traffic light becomes red.

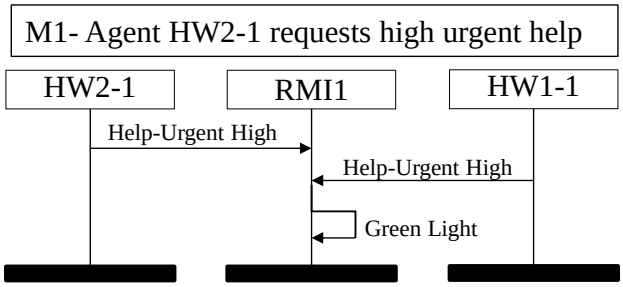

Figure 52: HW2 agent requests for a high urgent help [C1]

The high level MSC of the system is shown in Figure 54. The MSC M1 starts and then the second MSC can execute. There is a path from M2 to M1.

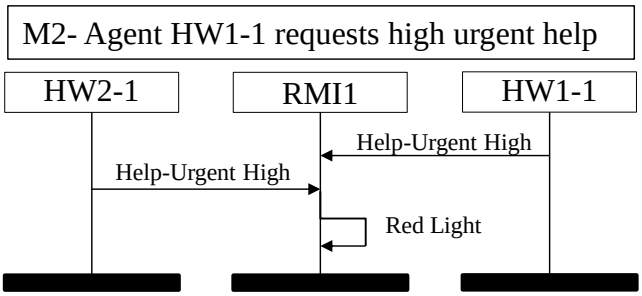

Figure 53: HW1 agent requests for a high urgent help [C1]

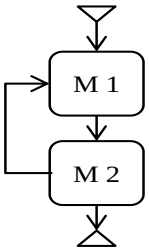

Figure 54: High level MSC of congestion control system [C1]

The asynchronous concatenation of MSCs means that in a hMSC the processes can proceed to the next MSC with different timings [162]. Consider two MSCs $\mathcal{M}_1, \mathcal{M}_2 \in \boldsymbol{\mathcal{M}}$ with processes $p, q \in P_{\boldsymbol{\mathcal{M}}}$and $\mathcal{M}_1$ precedes $\mathcal{M}_2$ in the hMSC $\mathcal{G}$. Asynchronous concatenation of MSCs is interpreted as the time that processes do not wait for other processes to accomplish their actions in one MSC. Therefore, for example, while process p is still executing the first MSC, process q may continue to $\mathcal{M}_2$. We refer to this as issue **I1** (SLIS-IV). Likewise, HW2-1 (in M1) can continue to

perform its actions in M2 and send a high urgent help request RMI1. However, agent HW1-1 is still executing M1. It results in repetition of the execution of the first scenario M1. As a result, the EB shown in Figure 55 (a) can happen. The same situation can happen for the other highway agent (HW1-1) in the second scenario M2 which results in the repetition of MSC M2. This is shown as another EB in Figure 55 (b).

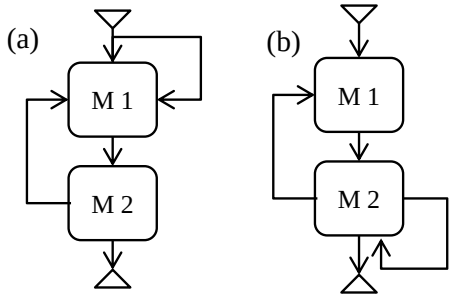

Figure 55: Two implied scenarios [C1]

The asynchronous concatenation of MSCs can also cause another problem that we refered as race conditions (CLEB-IV). To show this, consider the third scenario that is shown in Figure 56. In this scenario (M3), agent HW2-1 sends a “low urgent help” message to the ramp agent. RMI1 analyzes the situation by asking the other high way agent about the current situations in order to be able to decide and take a good action. If the urgency of HW1-1 is high the traffic light becomes red.

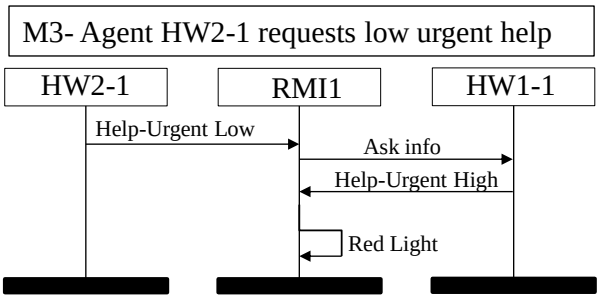

Figure 56: HW2 agent requests a low urgent help [C1]

The IS that can happen in this situation is shown in Figure 57. In this IS, HW2-1 can continue in M1 (i.e. send a request of high urgent). Since it is a race condition and there is no control over the messages that the ramp agent should receive, it can make the traffic light green instead of red which is defined for it in MSC M1.

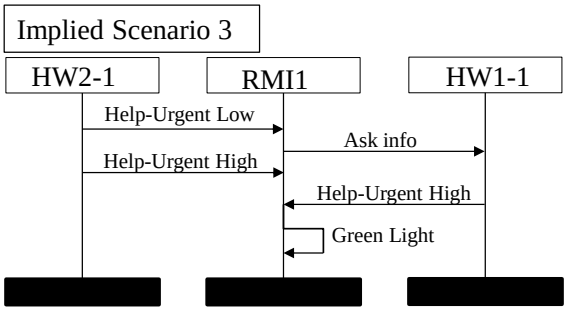

Figure 57: Implied scenario in congestion control system [C1]

We have defined active and passive processes previously. Based on the definitions of active and passive process, an active process p can lead to issue **I1** in all MSCs that it is active and there is no timing or control over these MSCs. Other than starting actions, when $\mathcal{G}_p$ is different from the hMSC, it can lead to an IS. This is the case for high way agents.

The cases for the race conditions are explained in previous chapters. It was explained that the order of receiving messages is changed which led to an implied scenario.

7.3 Summary

In this chapter we examined various case studies, both from the literature and some new ones. These case studies are Fleet management system, Greenhouse control system, Online auction system, and Traffic control system. Some of these systems are using MAS and the control is given to various agents. Through the case studies, we have explained how to model and analyze the system in order to find the EB/IS. Explanations on fixing the detected issues are also provided.

Chapter Eight: **Evaluation and results**

8.1 Introduction

In this section, we explain the results of our work. From the existing approaches in the literature, we have chosen the application of our methodology to various case studies. Also, for the evaluation part, we have considered some criteria that evaluates different aspects of our work. We report the results in multiple sections as follows:

1. We have chosen some of the works that have the greatest number of case studies, including new and cases from other works. We apply our modeling to these cases other than the ones we explained in the previous chapter.
2. The results of different steps of our work including the pre-processing phase.
3. The results of detecting EB/IS with applying data mining techniques for the functions in the detection methodology. We consider this part as an extension of our work that shows possible solutions for the detection of emergent behaviors.

The results show that our modeling covers all the implied scenarios that other works can detect. Moreover, since our technique models the interaction information for the processes, it can detect new emergent behaviors that cannot be detected by other approaches [150]. Also, in our modeling, we use a high level structure for each process which is a new approach. This high level structure is used in analyzing the problems that can happen in the system and also providing reasoning on the detected issues.

8.2 Evaluation and results

The evaluation of finding EB/IS is a challenging area, since there is no specific benchmark or data set for testing the methodologies. There are some researches which are considered as the main works in this field. In these works, the evaluation is accomplished by applying the methodology

on various case studies and try to find new EB/IS or false positives from the case studies in other works. In works that are focusing on the theoretical parts, the mathematical proofs are presented. In this work, we choose the first approach and focus on detecting the EB/IS from the case studies in the literature, as well as new ones with our methodology. In this process, our focus for the evaluation are the following criteria:

- Detecting number of existing problems in a case study to find out if all the existing EB/IS can be detected in a specific case study.
- Evaluating the detected EB/IS as false positive (FP) or true positive (TP), indicating whether the detected EB/IS are actually an EB/IS.
- The ability to detect problems in different communication styles (synchronous vs asynchronous).
- The capability in detecting the origins of the problem.
- The ability to develop solution repositories for the detected EB/IS.

For this thesis, we have chosen a set of case studies from a wide range of applications. These case studies are either new case studies or the ones that exist in the literature. Specifically the cases from the work of Song el at. (2009-2011) [37], Uchitel et al. (2003) [40], and Kumar (2010) [41] are chosen. These researches have a combination of theoretical and practical work, or their work includes critics on the flaws of the other ones. Therefore, it provides a wide range of criteria for comparison of our methodology to the others. These include the famous ATM machine [4], Boiler Control System [83], Automated Mine-Sweeping Robot [163], Semantic Search System [164], Internet Filtering System [165], Automated Manufacturing System [12], and Real-Time Fleet Management System [42].

8.2.1 Implementation

We have implemented the behavioral modeling and the detection algorithms in a simple tool called Eagle. At this stage, the tool works with command line and no GUI is provided. For the implementation, we have used C# and Weka API. The architecture of Eagle is shown in Figure 58.

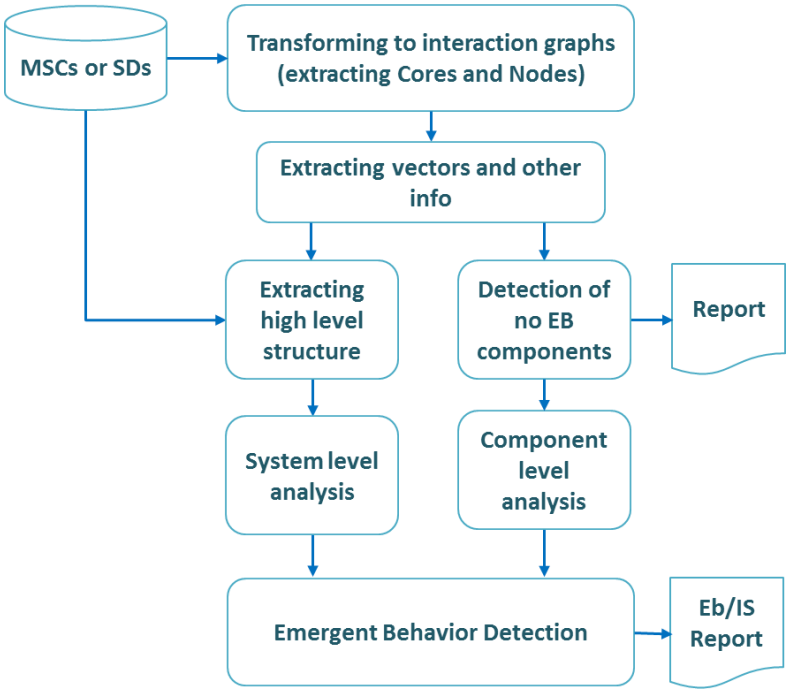

Figure 58. Eagle architecture and the main modules

As the inputs, it can accept the sequence diagrams from Microsoft Visio 2013 and the ITU-T message sequence charts as the text command. There are a lot of tools provided for MSCs that provide a text based definition for the MSCs. The inputs are then analyzed and modeled into the interaction graphs based on the definitions of Cores and Nodes. The advantage of having separate information in Cores and Nodes is to model the system once, and use the required information for component and system level analyses separately. Finding the shared states is with data mining techniques. For this part, we use message labels as the strings and use the API of Weka to analyze and extract the shared states. Finding the conditions that satisfy each class of EB/IS is based on

the algorithms provided previously. The only point that is worth mentioning here is that we analyze each class of EB/IS separately. Therefore, the user can choose which specific type must be checked. The results of the analysis is reported to the user through a text file, indicating the following information:

- The existence of EB/IS.
- The specific scenarios that lead to an issue.
- The specific component/components and the messages in the scenarios that lead to a problem (the exact point of the occurrence of EB/IS).
- The recommendation on how to fix the problem.

8.2.2 Implied scenario detection

In Table 2 the above mentioned criteria for the case studies that we have discussed in the previous chapter are shown. For each of the case studies, the number of scenarios indicated and the number of detected EB/IS after analyzing these scenarios (MSCs/hMSCs) are included as well. For all of the case studies, the reasons of having an EB/IS are identified in the detection process. Similarly, the recommendations on preventing the detected EB/IS are indicated. These two options become possible due to our approach for the detection of EB/IS which is using the EB Catalogue. As a result, the specific conditions that we check to know whether an EB/IS exist in the system, are directions on the origins of the problem and can lead to developing directions to prevent the occurrence of the detected issues. In this table, there is a column for FP/TP that shows whether the EB/IS found with our methodology are truly implied scenarios. For some works, it is claimed that their methodologies find FP or TN, i.e. finding some scenarios as implied scenarios which is wrong or they do not find some implied scenarios. This is due to the methodology used for behavioral modeling and issues like overgeneralization of the state machines. For our methodology, this case

is not true. In other words, we find all the implied scenarios that are not in the list of MSC specifications of the system and check them against the current behaviors. The list of behaviors that are not included in the current specifications of the system are announced as EB/IS. Our methodology will not provide new behaviors because we generate interaction graphs associated to each of the MSCs/SDs and avoid connections in unnecessary cases. Hence, the connections are only for some parts of the system level analysis, and are exactly as it is shown in the visual form of the MSCs/SDs in their high level structure (hMSC). This prevents in announcing scenarios as implied scenarios while they are in the list of SDs/MSCs of the system (avoiding FP). Moreover, the implied scenarios that we define and detect here are the ones that come from specific conditions (defined in previous sections) in the EB Catalogue. The main reasons of having EB/IS are shared states or active processes. In our methodology, we investigate all of the shared states for all processes, and also all of the active processes. Therefore, we are able to find all the implied scenarios (TP) and avoid having extra ones (FP) or ignoring the ones that exist (FN).

Lemma. All of implied scenarios that exist in the system can be detected with our methodology.

Proof. Based on the EB Catalogue, the main reason of having EB/IS in the system is having shared states or shared interactions among various scenarios and components of the system. Suppose that there are some scenarios that are implied scenarios and are not detected with our methodology. This implies that there are some shared states or shared interactions among various components of the system that are not analyzed with our technique. Based on the definitions in previous sections, this is not true; since we analyze all shared states among all scenarios of the system. As a result, all the implied scenarios are detected with our methodology. □

Lemma. All the detected EB/IS are found correctly, meaning that there is no scenario that is announced as implied scenario but it is in the system specification.

Proof. Suppose that there is a scenario which is wrongly detected as an implied scenario. It means that either the scenario was not in the system specifications at first, or it was in the system specifications and was announced as a wrong scenario. The former case means that the detected scenario is an implied scenario, which is not the case. The latter means that the scenario was not among the specifications of the system to be checked against, which is not true. As a result, all the scenarios detected as implied scenarios are truly tagged as an implied scenario (TP). □

Table 2. Evaluation criteria in studied cases

Case studies/ Criteria	# of scenarios	# of EB/IS detected	Detection of all of the EB/IS	FP/TP/FN	Syn/ Asyn	Reason	Solution
Online auction system	7	3	Yes	TP	Syn.	Yes	Yes
Traffic control system	4	2	Yes	TP	Syn.	Yes	Yes
Fleet management system	4	1	Yes	TP	Syn.	Yes	Yes
Greenhouse system	6	1	Yes	TP	Syn.	Yes	Yes
Mine sweeper robot	5	2	Yes	TP	Syn.	Yes	Yes
Boiler control system	5	4	Yes	TP	Asyn.	Yes	Yes

The criteria on detecting all of the existing EB/IS in Table 2 is also based on what exist in the literature. This column also shows that whether all of the implied scenarios reported in other works for a specific case study are detected with our methodology as well. Other than the proofs mentioned above, we have checked the implied scenarios detected in our work with the implied scenarios mentioned in the literature. The results show that we are able to find all the implied scenarios that other researches can find for each case study.

As an example for this part, we refer to the Boiler Control system. We have identified three of the implied scenarios that can exist in this system previously. In our model, we use the high level structure $\mathcal{G}_p$ for each of the processes in the system. The analysis of this structure is valuable in many aspects and can reveal a broad range of issues that can happen in a system. Examples of the analysis of this structure are explained previously (see sections 6.3.1.3.2. and 6.3.2.4.2.2 and 7.2.4).

To verify our work, we have worked on a wide range of case studies including the fleet management and other cases named above. For all of these systems, we have detected all the implied scenarios that exist in these systems, compared to the approaches other researchers have used. Moreover, with our methodology, some new implied scenarios were found that was missing in the existing approaches. One of the examples of the new implied scenarios that other works cannot detect is the specific type of EB, CLEB-II. We have also provided a comparison on some main researches providing information on the type of the detected problems, modeling used in their work, and whether they have provided solutions for the detected issues. We have also included a column on their ability to detect CLEB-II. The results are reported in Table 3 which we previously published in [153].

Table 3. Comparison of research on implied scenario detection and modeling approaches [153]

Research	year	Type of detected error	Modeling	Suggest solutions	Detection of CLEB-II
Ben	1998	Process divergence, Non-local choice	Modeling NL choice	x	x
Alur	2003	Deadlock, race condition, timing	State Machine	x	x
Whittle	2001	Conflicting behaviors	State Charts	x	x
Uchitel	2003	Implied Scenario	FSP + LTS	x	x
Muccini	2003	Non-local branching choice	Modeling NL choice	x	x
Song	2011	Implied scenario	Graph comparison	Provides reasons	x
Kumar	2000-2010	Implied scenario	State Machine	x	x
Mousavi	2009	Implied scenario	State Machine	x	x
Our work	2016	Asyn. concatenation, Various Implied scenarios	Interaction Modeling	Yes	Yes

8.2.3 Pre-processing phase

One of the contributions of our work is adding a pre-processing phase in order to reduce the number of components that must be analyzed for the existence of EB in their behavior. Previously, we have used the Online Auction System as a case study that shows how our algorithms can detect agents that are not candidates to produce an emergent behavior in the component level. We have applied our algorithm to three case studies from the literature [163-165]. The results are summarized in Table 4 which we previously published in [147]. The reduction of the number of components becomes more valuable when the number of components in the system are large.

Table 4. Summarizing the results of applying the pre-processing algorithm in different examples [147]

Researches	Detected agents that will not cause emergent behavior (Reduction in FSM)
Auction system [147]	**33%**
Semantic search system [164]	**40%**
Mine sweeper robot [163]	**40%**
Internet filtering system [165]	**0 %**

The system in [164] includes five components for a semantic search system: user, query handler, concept learner, local repository and peer finder. Two of them (user agent and peer finder) do not need synthetization of behavioral modeling based on our algorithm. Calculating the number of components and scenarios that require behavioral modeling, it means 40% reduction in constructing the FSMs. The system in [163] is a mine sweeping robot and the results is also the same for this system. The other example is an internet filtering system [165]. For this system, our

method could not find any safe components. Applying our method in this phase does not always result in the reduction in the number of components that require behavioral modeling. This is based on the system specifications and whether they satisfy the conditions we named previously for having safe components.

To evaluate these results, we checked the implied scenarios that were reported in each of these cases and none of them included implied scenarios that were related to the components that we reported as safe components in our algorithm.

8.2.4 Finding neutral nodes in a network of software agents

The approach we have used for the pre-processing phase can be used to detect neutral nodes in a social network of software agents. For this case, we cluster the send and receive vectors of the agents in all the diagrams in which they have some collaboration. By clustering these vectors, we can be sure that the agents that remain in the same cluster, always have similar send and receive actions. In other words, when the number of their cluster does not change, their functionality is the same. Therefore, they are considered as neutral agents. We have published the results of this work in [166].

Finding neutral nodes in a network of software agents can be useful in various applications of agent based modeling. The method analyzes the inter- and intra- interactions of the software agents. We have reported the results of this section previously in [149].

8.3 Summary

This chapter includes the results and evaluations of our work. We have used multiple case studies to evaluate our work. This is the current trend that is used in the literature for this type of research. To be more specific, we have determined some criteria to compare our work with the others. Also, one important factor in this comparison is to find whether all the implied scenarios

which are detected by other methodologies can be detected with our work. The results show that our methodology is able to find all of these implied scenarios, as well as new ones. The proofs for completeness of our methodology is also provided in this chapter. The results and evaluations are reported in separate tables in this chapter. The system architecture for our tool called Eagle is also explained in this chapter. The other sections in detail provide the results of our work in finding the implied scenarios and the pre-processing phase.

Chapter Nine: **Discussions**

9.1 Discussions

In this part, we would like to discuss some special aspects of our work and the motivations that led to this work at first, and then discuss the results. In the process of behavioral synthesis with finite state machines (FSM) [4, 92], two categories of emergent behaviors are introduced to the system: 1- Overgeneralization, which is caused by generalizing all the FSM of one agent into one FSM. 2- The unexpected behaviors caused by incompleteness in specification of agents [167] (only in Multiagent systems). In these works, other than the complexity of the behavioral modeling process, new problems are inserted into the system that should be detected. These are new issues, other than the ones that may exist due to the design flaws that should be detected. Moreover, the internal behavior of the components is required in this modeling. In other words, the events and what triggers a change in the event of a component depends on the messages. This is true for labelled transition systems and some other works, where the dependency of the messages should be considered as causal dependencies or pre- and post- conditions between the messages. These processes make the modeling and the foundation of the methodology dependent on humans. In the era of software automation, we meant to find a methodology to make part of this field fully automated. Our main question that we tried to answer in this thesis was: Is there a methodology that can find the EB/IS in the system without depending on subject experts and therefore makes the whole process of analyzing the UML sequence diagrams and message sequence charts from the system designs fully automatic? As explained previously, in this process we accomplished some important results such as making a catalogue of emergent behaviors and implied scenarios based on the origins of the problem.

A big discussion can be the use of behavioral modeling with states machines in this work. For this part, we try to find a different modeling methodology to prevent some challenges of behavioral modeling with state machines. In our modeling, we generate the interaction graphs from the components' specifications and avoid creating new graphs to combine the similar (shared) states of their behavior. As a result, our modeling does not bring or introduce overgeneralization or extra behavior to the behavioral model of the processes and the system. Another point of discussion is that we preserve the information about the interaction of software components in our modeling from the first part. Consequently, the interaction graphs can be used for any kind of analysis and there is no need to generate a separate graph that shows their interactions. This is closer to their implementation. The information that we model is: who they send a message to, message labels, and from which components they can receive a message, in each scenario. As explained before, this modeling has helped in the introduction of the implied scenario that is caused by some assumptions that are different in the designs of agents' communications and in the implementation of the system. These new implied scenarios are extracted from the interactions of the components, other than just considering the states or events on each process (agent). The existing works are not able to detect this implied scenario, because they do not consider the agents' communications.

Another challenge that we have found in many of the existing works is the assumption of having the behavioral models in state machines or developing algorithms based on the state machines and the usage of labelled transition systems without actually explaining the ***"how to"*** steps. For example, in [42], for each agent the concurrent automata is constructed, and the Labelled Transition Systems (LTS) is obtained. Then for the detection of the EB/IS, the safe realizability of the system should be determined. Other than the complex process of behavioral modeling and

constructing the LTS, there is no specific algorithm determined for obtaining LTS for each agent. Moreover, the algorithm for safe realizability is not devised. These issues exist for many of the works in the literature, since many works focus on the theoretical part of the IS detection and only include the definitions for each of these concepts. Therefore, in our approach, we have tried to develop the main steps for finding the EB/IS in the system. For each part, we have devised the general algorithms. In each of the algorithms, multiple functions are used that might have different implementations. For the main part of these functions (finding the process's shared states) and the conditions that lead to EB/IS, we have devised detection methods. One approach that we have chosen is the usage of data mining techniques.

The other discussable point in our work is the detection of EB/IS versus developing solutions for the detected problem. To the best of our knowledge, the only work that develops general solutions (not application specific) is our work. In many works, some suggestions are provided on specific examples which cannot be expanded to other applications. The reason that we are able to suggest general solutions is that we have focused on the origins of the problem that can exist in component or system level. We have classified the common EB/IS based on the reasons that might occur. Accordingly, we have developed our detection methodology and modeling based on the origins of EB/IS. As a consequence, we can find the reason and the exact point (event of a process and scenario) that the EB/IS exist, and we are able to point that problem and develop solutions. In the literature, the implied scenario is shown as an error that exists in the system. However, the problematic points (events, states) are not shown. There is one other approach that finds the points that an implied scenario can occur in an MSC [37]. The advantage of our modeling over this work is that, we detect the problematic points and indicate the probable causes for those points. Besides, we go one step further and suggest solutions based on the detected reasons. Furthermore, by

classifying the EB/IS in various classes, based on what the possible causes for each class are, we can use different information in the analysis of each class of implied scenarios.

The specific aspects of our work that differentiates it from other researches are:

1. Looking at the system as a black box. Consequently, in terms of Multiagent Systems, there is no need to know the internal knowledge space of the agents; and in terms of differences with the existing works, there is no need for identifying the conditions or causalities between the messages.
2. Looking for a general solutions and not being application specific. Therefore, our methodology is not dependent on the content of messages. Based on the results of the problems that we have categorized in the EB Catalogue, we find the arising EB/IS in the system. The methodology uses a labeling phase at first to change the message contents to message labels and then uses these labels only for the detection of EB/IS.
3. Adopt modeling the system by using the concepts from state machines and social network analysis and the interactions among components. In this modeling, we use the message labels that components send or receive as the nodes of the interaction graphs and connect the nodes to each other regarding the SD/MSC specifications. The states and the triggering events (messages) are combined into the nodes and only the links to the next states are preserved.
4. Presenting the EB Catalogues. This catalogue is based on the origins of the problems in both component and system levels. Based on this catalogue, we have developed general detection algorithms and solution repositories. The detection of the causes of the EB/IS and presenting solutions for each type of EB/IS are also among other novel aspects of this catalogue.

5. Being able to use data mining (DM) techniques to detect EB/IS in the system is another aspect that differentiates our methodology with other works. Modeling the system with the interaction graphs and the reasons provided in the EB Catalogue, provides the basic circumstances to use DM. One of the advantages of using DM in this process is decreasing the number of required comparisons to find the shared states, and its ability to predict new paths. This is elaborated more in the future areas of this work.

The other point that we like to discuss in this section is the test bed we created but were not able to use for this work. We set up a Linux cluster and executed some open source MPI programs on this cluster. Our main focus in this process was whether we are able to use the communications between the processes and detect EB/IS. However, the feasibility analysis of this phase resulted in not suing this cluster as a test bed, since it would have some contradictions with our assumption to consider the processes as black boxes. Analyzing the communication between the processes in MPI requires to create a tool that controls or changes the execution of different MPI functions. Not only this made it unsuitable for our purpose, it is far beyond the scope of this work and requires to be accomplished as a separate Masters or PhD project.

Model checking that is used for our area of work, uses two approaches: modeling the system to check it against a specific property, behavioral modeling to find EB/IS. We refer to the second category of the works. Therefore, the correctness of our methodology is interpreted as finding the EB/IS which is evaluated and the results are reported in other sections.

9.2 Conclusion

In this thesis, we have categorized the common EB/IS that exist in MAS and DSS. This catalogue is based on our research and the problems that are described in the literature. The strategy that we have used in this classification is focusing on the origins of the problems. Other than the existing

works, we have introduced a new implied scenario that can happen in MAS due to specific conditions that exist for the communication of various agents in the system. This issue can be analyzed by our methodology, because the information about the agents' interactions are preserved in our modeling. The other feature of the EB Catalogue is classifying the issues based on the EB/IS that can happen in component and system level. Each of these classes have some sub-classes. These EB/IS types are not explicitly mentioned or reviewed in the literature. The other contribution of our work is looking at DSS/MAS as black boxes and try to find a methodology that certifies their behavior without considering the details of the components/agents. In this approach, we also tried to answer this research question: Is there a general approach that can detect EB/IS without human interference and is fully automated? Is there an approach that considers the interactions among software agents in order to certify their behavior in early phases? The answer of these questions led us to develop a new methodology for the modeling and detection of emergent behaviors and implied scenarios based on the combination of concepts from state machines and social network analysis. In this process, we have modeled the system and the behavior of the components with interaction graphs. This modeling is based on the system specifications defined with MSCs or SDs.

One other part of our work is dedicated to develop detection methodology for each class of EB/IS in the catalogue. For each class, specific conditions that can lead to EB/IS are identified. Referring to these, by detecting the exact point of the problem, we are able to suggest general solutions for the detected issues. All of these steps are among contributions of our work. We have provided examples for all of these steps in different chapters of this thesis.

In the literature, there is no baseline for the comparison of different works. The evaluation in this field for practical works is done by using various case studies from the literature. We have

chosen the same evaluation technique in this thesis. To be more precise, we have defined some criteria for this evaluation and reported our results based on those criteria. We have applied our technique to several cases from the literature, as well as some new ones. Based on the results, our methodology is able to detect all the implied scenarios as well as new ones that other approaches cannot detect or are not able to find its reasons.

We have also devised methods to find the components that will not cause emergent behavior in a distributed system both in synchronous and asynchronous communication styles. Those components/agents can be eliminated in the behavioral synthesis phase. Many algorithms that use behavioral modeling for the detection of emergent behaviors have a computation complexity. A direct advantage of this method is to help reduce the computation complexity of the algorithms. This method is generic and can be considered as a preprocessing phase to eliminate the components that are not a point of emerging new behavior in component level. It is worth mentioning that even these components should be analyzed for the system level analysis. Also, the algorithm in some cases does not find any components, due to the system specifications.

Other than modeling with interaction graphs, we can model the system with interaction matrices as explained in the previous chapter. One advantage of these matrices is having various matrices for send and receive messages. In the literature, it is assumed that the messages are received at the same time they are sent. This assumption is not what happens in the real world. The introduction of different matrices for send and receive events can help investigate the effect of service degradation in the case of lost or corrupt messages. One issue in the usage of these matrices is having n^2 objects where most of them are zero entities. This can be solved by applying different techniques for the sparse matrix analysis.

All the above mentioned works is implemented in a tool that works with command line. The architecture of the system is provided in this thesis. The current state of the system is to improve the background algorithms. The future plans are to provide a user friendly GUI and improve the visualization parts to include everything in the tool itself. One other plan is to write a plug-in for Eclipse and Visio.

The limitations of our work are requiring the specifications of the system in the form of MSC or SD. Also, this modeling and methodology only works for the systems that lack a centralized controller and multiple components are interacting to each other. For example, the methodology is not suitable for MapReduce and Hadoop, since there is a master node that controls the distribution of the job. The current methodology works with atomic interactions.

9.3 Extension areas

We have done a few other researches for various parts of this work, either as creating a test bed to analyze the feasibility of other approaches to be used in our work, or the applicability of this work in other areas. Throughout this process, we have found the following research areas as valuable works to be completed in the future.

1. Using data mining and sequence mining for the detection of shared states. This will help in analyzing the sequence of message labels (the language in state machines) in all the scenarios of the system. The other application in this area is mining the current scenarios of the system to classify the behavior of specific components. This can be used in various analyses such as load balancing and performance of specific components.
2. Visualization of the results for the designers as part of the tool for certifying the behavior of the system.

3. Applying this technique for the detection of problems in parallel programming such as MPI. Part of our research was dedicated to analyzing the feasibility of using MPI programs as a test bed in our work. In this process, we have found that the design of parallel programs can be challenging and requires expertise in this area. Finding the EB/IS in this field can be interpreted as analyzing the behavior of different processes and how they see the states of the system. Therefore, having a tool that can help in the design of these systems can be valuable for the developers. Specifically talking about MPI programs, this requires focusing on the specific functions that are used in MPI and are provided for the communications of the processes (Such as Broadcast, Send, Blocking Receive, etc.). For this reason, the tool should work with the MPI library as well.
4. Using interaction graphs to find the new paths in the automated generated code. This will make the application of EB/IS detection close to the implementation. We consider this work as the major future work of this thesis. The results provide a cost effective solution to DSS verification which appeals to a large audience (medium/large companies), as well as academia/industry in the distributed computation fields.

References

1. Haugen, and R.K. Runde, *Enhancing UML to Formalize the FIPA Agent Interaction Protocol, Agent-Based Technologies and Applications for Enterprise Interoperability*, K. Fischer, et al., Editors. 2009, Springer Berlin Heidelberg. p. 154-173.
2. Krüger, I.H., *Distributed System Design with Message Sequence Charts*, in *Institute of computer science*. 2000, Technical University of Munich. p. 386.
3. Genest, B., A. Muscholl, and D. Peled, *Message Sequence Charts Lectures on Concurrency and Petri Nets*, J. Desel, W. Reisig, and G. Rozenberg, Editors. 2004, Springer Berlin / Heidelberg. p. 103-121.
4. Mousavi, A., *Inference of Emergent Behaviours of Scenario-Based Specifications*, in *Department of Electrical and Computer Engineering*. 2009, University of Calgary: Calgary. p. 160.
5. Briand, L.C., *Software Verification — A Scalable, Model-Driven, Empirically Grounded Approach.* 2010: p. 415-442.
6. Pierce, P. *Software verification and validation*. in *Northcon/96*. 1996.
7. Yamada, S., *An Experimental Study of Human Factors in Software Reliability Based on a Quality Engineering Approach*, in *Springer Handbook of Engineering Statistics*, H. Pham, Editor. 2006, Springer London. p. 497-506.
8. Broy, M., *Seamless Method- and Model-based Software and Systems Engineering.* 2011: p. 33-47.
9. Alur, R., K. Etessami, and M. Yannakakis, *Inference of Message Sequence Charts.* IEEE Trans. Softw. Eng., 2003. **29**(7): p. 623-633.
10. Bultan, T. and C. Heitmeyer, *Applying infinite state model checking and other analysis techniques to tabular requirements specifications of safety-critical systems.* Design Automation for Embedded Systems, 2008. **12**(1-2): p. 97-137.
11. Eldin, M.N., A. Kamel, and O. Hegazy. *Capture-recapture techniques in software verification*. in *Computer Engineering & Systems, 2008. ICCES 2008. International Conference on*. 2008.
12. Moshirpour, M., *Model-Based Detection of Emergent Behavior in Distributed and Multi-Agent Systems from Component Level Perspective*, in *DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING*. 2011, University of Calgary: Calgary. p. 109.
13. Moshirpour, M., A. Mousavi, and B.H. Far. *A Technique and a Tool to Detect Emergent Behavior of Distributed Systems Using Scenario-Based Specifications*. in *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*. 2010.
14. Alur, R. and M. Yannakakis, *Model Checking of Message Sequence Charts*, in *CONCUR'99 Concurrency Theory*, J.M. Baeten and S. Mauw, Editors. 1999, Springer Berlin Heidelberg. p. 114-129.
15. Ben-Abdallah, H. and S. Leue, *Mesa: Support for scenario-based design of concurrent systems*, in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Steffen, Editor. 1998, Springer Berlin Heidelberg. p. 118-135.
16. Uchitel, S. and J. Kramer, *A workbench for synthesising behaviour models from scenarios*, in *Proceedings of the 23rd International Conference on Software Engineering*. 2001, IEEE Computer Society: Toronto, Ontario, Canada. p. 188-197.

17. Muccini, H., *Detecting Implied Scenarios Analyzing Non-local Branching Choices*, in *Fundamental Approaches to Software Engineering*, M. Pezzè, Editor. 2003, Springer Berlin Heidelberg. p. 372-386.
18. Mitchell, B., *Characterizing Communication Channel Deadlocks in Sequence Diagrams.* Software Engineering, IEEE Transactions on, 2008. **34**(3): p. 305-320.
19. Alur, R. and M. Yannakakis, *Model checking of hierarchical state machines.* ACM Trans. Program. Lang. Syst., 2001. **23**(3): p. 273-303.
20. Ammar, K., L. Pullum, and B. Taylor, *Augmentation of Current Verification and Validation Practices*, in *Methods and Procedures for the Verification and Validation of Artificial Neural Networks*. 2006, Springer US. p. 13-31.
21. Gluck, P.R. and G.J. Holzmann. *Using SPIN model checking for flight software verification*. in *Aerospace Conference Proceedings, 2002. IEEE*. 2002.
22. Sanchez, E., G. Squillero, and A. Tonda, *Automatic Software Verification*, in *Industrial Applications of Evolutionary Algorithms*. 2012, Springer Berlin Heidelberg. p. 17-30.
23. Pelliccione, P., et al., *TeStor: Deriving Test Sequences from Model-Based Specifications*, in *Component-Based Software Engineering*, G. Heineman, et al., Editors. 2005, Springer Berlin Heidelberg. p. 267-282.
24. Damiani, E., C. Ardagna, and N. El Ioini, *Formal methods for software verification*, in *Open Source Systems Security Certification*. 2009, Springer US. p. 1-26.
25. Hinchey, M., J. Bowen, and C. Rouff, *Introduction to Formal Methods*, in *Agent Technology from a Formal Perspective*, C. Rouff, et al., Editors. 2006, Springer London. p. 25-64.
26. Wang, C., et al., *Dynamic Model Checking with Property Driven Pruning to Detect Race Conditions*, in *Automated Technology for Verification and Analysis*, S. Cha, et al., Editors. 2008, Springer Berlin Heidelberg. p. 126-140.
27. Bensalem, S. and K. Havelund, *Dynamic Deadlock Analysis of Multi-threaded Programs*, in *Hardware and Software, Verification and Testing*, S. Ur, E. Bin, and Y. Wolfsthal, Editors. 2006, Springer Berlin Heidelberg. p. 208-223.
28. Bertolino, A., H. Muccini, and A. Polini, *Architectural Verification of Black-Box Component-Based Systems*, in *Rapid Integration of Software Engineering Techniques*, N. Guelfi and D. Buchs, Editors. 2007, Springer Berlin Heidelberg. p. 98-113.
29. Verhulst, E., G. Jong, and V. Mezhuyev, *An Industrial Case: Pitfalls and Benefits of Applying Formal Methods to the Development of a Network-Centric RTOS*, in *FM 2008: Formal Methods*, J. Cuellar, T. Maibaum, and K. Sere, Editors. 2008, Springer Berlin Heidelberg. p. 411-418.
30. Chaki, S., et al., *State/Event Software Verification for Branching-Time Specifications*, in *Integrated Formal Methods*, J. Romijn, G. Smith, and J. Pol, Editors. 2005, Springer Berlin Heidelberg. p. 53-69.
31. Quan, T.T., et al., *MAFSE: A Model-Based Framework for Software Verification.* 2010: p. 150-156.
32. Knight, J., *Challenges in the utilization of formal methods*, in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, A. Ravn and H. Rischel, Editors. 1998, Springer Berlin Heidelberg. p. 1-17.
33. Kneuper, R., *Limits of formal methods.* Formal Aspects of Computing, 1997. **9**(4): p. 379-394.

34. Iglesias, A., *Software Verification and Validation of Graphical Web Services in Digital 3D Worlds*, in *Communication and Networking*, D. Ślęzak, et al., Editors. 2009, Springer Berlin Heidelberg. p. 293-300.
35. Holzmann, G.J. and M.H. Smith, *An automated verification method for distributed systems software based on model extraction.* Software Engineering, IEEE Transactions on, 2002. **28**(4): p. 364-377.
36. Uchitel, S., *Partial Behaviour Modelling: Foundations for Incremental and Iterative Model-Based Software Engineering*, in *Formal Methods: Foundations and Applications*, M. Oliveira and J. Woodcock, Editors. 2009, Springer Berlin Heidelberg. p. 17-22.
37. Song, I.-G., et al., *An approach to identifying causes of implied scenarios using unenforceable orders.* Inf. Softw. Technol., 2011. **53**(6): p. 666-681.
38. Krüger, I., M. Meisinger, and M. Menarini, *Runtime Verification of Interactions: From MSCs to Aspects*, in *Runtime Verification*, O. Sokolsky and S. Taşıran, Editors. 2007, Springer Berlin Heidelberg. p. 63-74.
39. Haglich, P., C. Rouff, and L. Pullum, *Detecting Emergence in Social Networks*, in *Proceedings of the 2010 IEEE Second International Conference on Social Computing*. 2010, IEEE Computer Society. p. 693-696.
40. Uchitel, S., *Incremental Elaboration of Scenario-Based Specifications and Behaviour Models Using Implied Scenarios*, in *Department of Computing*. 2003, Imperial College of Science, Technology and Medicine, University of London. p. 173.
41. Chakraborty, J., D. D'Souza, and K. Narayan Kumar, *Analysing Message Sequence Graph Specifications*, in *Leveraging Applications of Formal Methods, Verification, and Validation*, T. Margaria and B. Steffen, Editors. 2010, Springer Berlin Heidelberg. p. 549-563.
42. Mireslami, S., *Verification of Multi-Agent Systems Using AUML Methodology*. 2013, University of Calgary.
43. Havelund, K. and A. Goldberg, *Verify Your Runs*, in *Verified Software: Theories, Tools, Experiments*, B. Meyer and J. Woodcock, Editors. 2008, Springer Berlin Heidelberg. p. 374-383.
44. Koo, S.R., H.S. Son, and P.H. Seong, *NuSEE: Nuclear Software Engineering Environment*, in *Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems*, P.H. Seong, Editor. 2009, Springer London. p. 121-135.
45. Bubel, R., R. Hähnle, and R. Ji, *Program Specialization via a Software Verification Tool*, in *Formal Methods for Components and Objects*, B. Aichernig, F. Boer, and M. Bonsangue, Editors. 2012, Springer Berlin Heidelberg. p. 80-101.
46. Ochoa, O., et al., *Towards a Tool for Generating Aspects from MEDL and PEDL Specifications for Runtime Verification*, in *Runtime Verification*, O. Sokolsky and S. Taşıran, Editors. 2007, Springer Berlin Heidelberg. p. 75-86.
47. Barringer, H., et al., *Rule Systems for Runtime Verification: A Short Tutorial*, in *Runtime Verification*, S. Bensalem and D. Peled, Editors. 2009, Springer Berlin Heidelberg. p. 1-24.
48. Roozbehani, M., E. Feron, and A. Megrestki, *Modeling, Optimization and Computation for Software Verification*, in *Hybrid Systems: Computation and Control*, M. Morari and L. Thiele, Editors. 2005, Springer Berlin Heidelberg. p. 606-622.

49. Alglave, J., et al., *Making Software Verification Tools Really Work*, in *Automated Technology for Verification and Analysis*, T. Bultan and P.-A. Hsiung, Editors. 2011, Springer Berlin Heidelberg. p. 28-42.
50. Broy, M., *Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones.* Innovations in Systems and Software Engineering, 2006. **3**(1): p. 75-102.
51. Fitzgerald, J.S., *Two industrial trials of formal specification*, in *Algebraic Methodology and Software Technology*, M. Wirsing and M. Nivat, Editors. 1996, Springer Berlin Heidelberg. p. 1-8.
52. Broy, M., *A Theory of System Interaction: Components, Interfaces, and Services*, in *Interactive Computation*, D. Goldin, S. Smolka, and P. Wegner, Editors. 2006, Springer Berlin Heidelberg. p. 41-96.
53. Alur, R., et al., *Hierarchical modeling and analysis of embedded systems.* Proceedings of the IEEE, 2003. **91**(1): p. 11-28.
54. Alur, R. *Formal verification of hybrid systems*. in *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*. 2011.
55. *The Verification and Validation Life Cycle*, in *Software Verification and Validation*. 2007, Springer US. p. 85-154.
56. *IEEE Standard for Software Verification and Validation.* IEEE Std. 1012-1998, 1998: p. i.
57. *IEEE Standard for Software Verification and Validation.* IEEE Std 1012-2004 (Revision of IEEE Std 1012-1998), 2005: p. 1-110.
58. *Managing Verification and Validation*, in *Software Verification and Validation*. 2007, Springer US. p. 7-83.
59. Spivey, J.M., *The Z Notation: A Reference Manual*. second ed. 1998: Programming Research Group, University of Oxford. 168.
60. Bowen, J. and V. Stavridou, *The industrial take-up of formal methods in safety-critical and other areas: A perspective*, in *FME '93: Industrial-Strength Formal Methods*, J.P. Woodcock and P. Larsen, Editors. 1993, Springer Berlin Heidelberg. p. 183-195.
61. Broy, M., *From "Formal Methods" to System Modeling*, in *Formal Methods and Hybrid Real-Time Systems*, C. Jones, Z. Liu, and J. Woodcock, Editors. 2007, Springer Berlin Heidelberg. p. 24-44.
62. Bustard, D. and A. Winstanley, *Making changes to formal specifications: Requirements and an example*, in *Software Engineering — ESEC '93*, I. Sommerville and M. Paul, Editors. 1993, Springer Berlin Heidelberg. p. 115-126.
63. ITU, T.S.S.O., *Formal description techniques (FDT) – Specification and Description Language (SDL)*. 1999, INTERNATIONAL TELECOMMUNICATION UNION.
64. Alur, R., et al., *Analysis of recursive state machines.* ACM Trans. Program. Lang. Syst., 2005. **27**(4): p. 786-818.
65. Alur, R. and D.L. Dill, *Automata For Modeling Real-Time Systems*, in *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*. 1990, Springer-Verlag. p. 322-335.
66. Group, O.M. *Unified Modeling Language*. Available from: http://www.omg.org/.
67. Kryvyi, S., O. Chugayenko, and L. Matveeva, *Exploring the properties of MSC documents by translating them into Petri nets.* Cybernetics and Systems Analysis, 2009. **45**(6): p. 997-1003.

68. Rockstrom, A. and R. Saracco, *SDL--CCITT Specification and Description Language.* Communications, IEEE Transactions on, 1982. **30**(6): p. 1310-1318.
69. Collet, P., et al., *Composite Contract Enforcement in Hierarchical Component Systems*, in *Software Composition*, M. Lumpe and W. Vanderperren, Editors. 2007, Springer Berlin Heidelberg. p. 18-33.
70. Alur, R. and D.L. Dill, *A theory of timed automata.* Theor. Comput. Sci., 1994. **126**(2): p. 183-235.
71. Holzmann, G.J., *The model checker SPIN.* Software Engineering, IEEE Transactions on, 1997. **23**(5): p. 279-295.
72. Chaki, S. and J. Ivers, *Software model checking without source code.* Innovations in Systems and Software Engineering, 2010. **6**(3): p. 233-242.
73. Broy, M., et al., *Software and System Modeling Based on a Unified Formal Semantics*, in *Requirements Targeting Software and Systems Engineering*, M. Broy and B. Rumpe, Editors. 1998, Springer Berlin Heidelberg. p. 43-68.
74. Holzmann, G.J., R. Joshi, and A. Groce, *Swarm Verification Techniques.* Software Engineering, IEEE Transactions on, 2011. **37**(6): p. 845-857.
75. Bartocci, E., R. DeFrancisco, and S.A. Smolka, *Towards a GPGPU-parallel SPIN model checker*, in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. 2014, ACM: San Jose, CA, USA. p. 87-96.
76. Holzmann, G.J., *Parallelizing the spin model checker*, in *Model Checking Software*. 2012, Springer. p. 155-171.
77. Zeller, A., *Mining models*, in *Model Checking Software*. 2012, Springer. p. 23-23.
78. Leue, S. and M.T. Befrouei, *Counterexample explanation by anomaly detection*. 2012: Springer.
79. Brat, G., et al. *Java PathFinder-second generation of a Java model checker*. in *In Proceedings of the Workshop on Advances in Verification*. 2000: Citeseer.
80. Ball, T., V. Levin, and S.K. Rajamani, *A decade of software model checking with SLAM.* Commun. ACM, 2011. **54**(7): p. 68-76.
81. Corbett, J.C., et al. *Bandera: extracting finite-state models from Java source code*. in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*. 2000.
82. Henzinger, T.A., et al., *Software Verification with BLAST*, in *Model Checking Software: 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings*, T. Ball and S.K. Rajamani, Editors. 2003, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 235-239.
83. Sousa, F.C.d., et al., *Detecting Implied Scenarios from Execution Traces*, in *Proceedings of the 14th Working Conference on Reverse Engineering*. 2007, IEEE Computer Society. p. 50-59.
84. Uchitel, S., J. Kramer, and J. Magee, *Negative scenarios for implied scenario elicitation*, in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. 2002, ACM: Charleston, South Carolina, USA. p. 109-118.
85. Adsul, B., et al., *Causal Closure for MSC Languages*, in *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, S. Sarukkai and S. Sen, Editors. 2005, Springer Berlin Heidelberg. p. 335-347.

86. Bhateja, P., et al., *Local Testing of Message Sequence Charts Is Difficult,* in *Fundamentals of Computation Theory*, E. Csuhaj-Varjú and Z. Ésik, Editors. 2007, Springer Berlin Heidelberg. p. 76-87.
87. Okamoto, M. and Y. Niitsu. *Designing an SCE for the advanced IN applying the service software verification method*. in *Global Telecommunications Conference, 1994. GLOBECOM '94. Communications: The Global Bridge., IEEE*. 1994.
88. Alur, R., K. Etessami, and M. Yannakakis, *Realizability and verification of MSC graphs.* Theor. Comput. Sci., 2005. **331**(1): p. 97-114.
89. Schumann, J. and J. Whittle, *Automatic Synthesis of Agent Designs in UML, Formal Approaches to Agent-Based Systems*, J. Rash, et al., Editors. 2001, Springer Berlin / Heidelberg. p. 148-162.
90. Whittle, J. and J. Schumann, *Scenario-Based Engineering of Multi-Agent Systems, Agent Technology from a Formal Perspective*, C. Rouff, et al., Editors. 2006, Springer London. p. 159-189.
91. Whittle, J., R. Kwan, and J. Saboo, *From scenarios to code: An air traffic control case study.* Software and Systems Modeling, 2005. **4**(1): p. 71-93.
92. Whittle, J. and J. Schumann, *Generating statechart designs from scenarios*, in *Proceedings of the 22nd international conference on Software engineering*. 2000, ACM: Limerick, Ireland. p. 314-323.
93. Ben-Abdallah, H. and S. Leue, *Syntactic detection of process divergence and non-local choice in message sequence charts*, in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Brinksma, Editor. 1997, Springer Berlin Heidelberg. p. 259-274.
94. Song, I.-G., S.-U. Jeon, and D.-H. Bae, *A Graph Based Approach to Detecting Causes of Implied Scenarios under the Asynchronous and Synchronous Communication Styles*, in *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*. 2009, IEEE Computer Society. p. 53-60.
95. Uchitel, S., J. Kramer, and J. Magee, *Implied Scenario Detection in the Presence of Behaviour Constraints.* Electronic Notes in Theoretical Computer Science, 2002. **65**(7): p. 65-84.
96. Uchitel, S., J. Kramer, and J. Magee, *Synthesis of Behavioral Models from Scenarios.* IEEE Trans. Softw. Eng., 2003. **29**(2): p. 99-115.
97. Letier, E., et al., *Deriving event-based transition systems from goal-oriented requirements models.* Automated Software Engineering, 2008. **15**(2): p. 175-206.
98. Letier, E., et al., *Monitoring and control in scenario-based requirements analysis*, in *Proceedings of the 27th international conference on Software engineering*. 2005, ACM: St. Louis, MO, USA. p. 382-391.
99. Magee, J., et al., *Graphical animation of behavior models*, in *Proceedings of the 22nd international conference on Software engineering*. 2000, ACM: Limerick, Ireland. p. 499-508.
100. Henriksen, J., et al., *Regular Collections of Message Sequence Charts*, in *Mathematical Foundations of Computer Science 2000*, M. Nielsen and B. Rovan, Editors. 2000, Springer Berlin Heidelberg. p. 405-414.
101. Mukund, M., K.N. Kumar, and M. Sohoni, *Synthesizing Distributed Finite-State Systems from MSCs*, in *CONCUR 2000 — Concurrency Theory*, C. Palamidessi, Editor. 2000, Springer Berlin Heidelberg. p. 521-535.

102. Rosson, M.B. and J.M. Carroll, *Scenario-based design*, in *The human-computer interaction handbook*, A.J. Julie and S. Andrew, Editors. 2003, L. Erlbaum Associates Inc. p. 1032-1050.
103. Carrol, J.M. *Five reasons for scenario-based design*. in *System Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*. 1999.
104. Lunjin, L. and K. Dae-Kyoo. *Required Behavior of Sequence Diagrams: Semantics and Refinement*. in *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*. 2011.
105. Genest, B. and A. Muscholl. *Message sequence charts: a survey*. in *Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on*. 2005.
106. Mani, N., V. Garousi, and B.H. Far. *Monitoring Multi-Agent Systems for deadlock detection based on UML models*. in *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*. 2008.
107. Broy, M. *The essence of message sequence charts*. in *Multimedia Software Engineering, 2000. Proceedings. International Symposium on*. 2000.
108. UNION), T.S.S.O.I.I.T., *SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS - Formal description techniques (FDT) – Message Sequence Chart*. 1999, ITU-T Recommendation Z.120. p. 136.
109. UNION), T.S.S.O.I.I.T., *SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS - Formal description techniques (FDT) – Message Sequence Chart* 2004, ITU-T Recommendation Z.120. p. 136.
110. Leucker, M., P. Madhusudan, and S. Mukhopadhyay, *Dynamic Message Sequence Charts*, *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science*, M. Agrawal and A. Seth, Editors. 2002, Springer Berlin / Heidelberg. p. 253-264.
111. Bikram, S. and R. Cleaveland, *Triggered Message Sequence Charts.* Software Engineering, IEEE Transactions on, 2006. **32**(8): p. 587-607.
112. Uchitel, S., J. Kramer, and J. Magee, *Detecting implied scenarios in message sequence chart specifications*, in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. 2001, ACM: Vienna, Austria. p. 74-82.
113. Aggarwal, C., *An Introduction to Social Network Data Analytics*, in *Social Network Data Analytics*, C.C. Aggarwal, Editor. 2011, Springer US. p. 1-15.
114. Sabater, J. and C. Sierra, *Reputation and social network analysis in multi-agent systems*, in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*. 2002, ACM: Bologna, Italy. p. 475-482.
115. Mislove, A.E., *Online social networks: Measurement, analysis, and applications to distributed information systems*. 2009, Rice University.
116. Song, H.H., et al., *Scalable proximity estimation and link prediction in online social networks*, in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. 2009, ACM: Chicago, Illinois, USA. p. 322-335.
117. Hanneman, R.A. and M. Riddle. *Introduction to social network methods*. [online text book] 2005; Available from: http://faculty.ucr.edu/~hanneman/nettext/C6_Working_with_data.html.

118. Sun, J. and J. Tang, *A Survey of Models and Algorithms for Social Influence Analysis*, in *Social Network Data Analytics*, C.C. Aggarwal, Editor. 2011, Springer US. p. 177-214.
119. Passmore, D.L., *Social Network Analysis: theory and applications*.
120. Gutiérrez, C., I. García-Magariño, and R. Fuentes-Fernández, *Detection of undesirable communication patterns in multi-agent systems.* Engineering Applications of Artificial Intelligence, 2011. **24**(1): p. 103-116.
121. Gutierrez, C., et al., *Revealing bullying patterns in multi-agent systems.* J. Syst. Softw., 2011. **84**(9): p. 1563-1575.
122. Gutiérez, C., I. García-Magariño, and J. Gómez-Sanz, *Evaluation of Multi-Agent System Communication in INGENIAS*, in *Bio-Inspired Systems: Computational and Ambient Intelligence*, J. Cabestany, et al., Editors. 2009, Springer Berlin Heidelberg. p. 619-626.
123. Botía, J., J. Hernansáez, and A. Gómez-Skarmeta, *On the Application of Clustering Techniques to Support Debugging Large-Scale Multi-Agent Systems*, in *Programming Multi-Agent Systems*, R. Bordini, et al., Editors. 2007, Springer Berlin Heidelberg. p. 217-227.
124. Botía, J., J. Gómez-Sanz, and J. Pavón, *Intelligent Data Analysis for the Verification of Multi-Agent Systems Interactions*, in *Intelligent Data Engineering and Automated Learning – IDEAL 2006*, E. Corchado, et al., Editors. 2006, Springer Berlin Heidelberg. p. 1207-1214.
125. Wan, W., J. Bentahar, and A. Ben Hamza, *Model Checking Epistemic and Probabilistic Properties of Multi-agent Systems*, in *Modern Approaches in Applied Intelligence*, K. Mehrotra, et al., Editors. 2011, Springer Berlin Heidelberg. p. 68-78.
126. Benerecetti, M. and A. Cimatti, *Validation of Multiagent Systems by Symbolic Model Checking*, in *Agent-Oriented Software Engineering III*, F. Giunchiglia, J. Odell, and G. Weiß, Editors. 2003, Springer Berlin Heidelberg. p. 32-46.
127. Cohen, M., et al., *Abstraction in model checking multi-agent systems*, in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*. 2009, International Foundation for Autonomous Agents and Multiagent Systems: Budapest, Hungary. p. 945-952.
128. Lomuscio, A., H. Qu, and F. Russo, *Automatic Data-Abstraction in Model Checking Multi-Agent Systems*, in *Model Checking and Artificial Intelligence*, R. Meyden and J.-G. Smaus, Editors. 2011, Springer Berlin Heidelberg. p. 52-68.
129. Bordini, R.H., et al., *Model checking rational agents.* Intelligent Systems, IEEE, 2004. **19**(5): p. 46-52.
130. Wooldridge, M., et al., *Model checking multi-agent systems with MABLE*, in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*. 2002, ACM: Bologna, Italy. p. 952-959.
131. Bordini, R., et al., *Directions for Agent Model Checking**, in *Specification and Verification of Multi-agent Systems*. 2010, Springer. p. 103-123.
132. Dastani, M., K.V. Hindriks, and J.-J. Meyer, *Specification and verification of multi-agent systems*. 2010: Springer Science & Business Media.
133. Bonakdarpour, B., et al. *From high-level component-based models to distributed implementations*. in *Proceedings of the tenth ACM international conference on Embedded software*. 2010: ACM.

134. Basu, A., M. Bozga, and J. Sifakis. *Modeling heterogeneous real-time components in BIP*. in *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*. 2006: Ieee.
135. Bonakdarpour, B., et al., *A framework for automated distributed implementation of component-based models*. Distributed Computing, 2012. **25**(5): p. 383-409.
136. Bonakdarpour, B., M. Bozga, and J. Quilbeuf, *Model-based implementation of distributed systems with priorities.* Design Automation for Embedded Systems, 2013. **17**(2): p. 251-276.
137. Arbab, F., *Reo: a channel-based coordination model for component composition.* Mathematical structures in computer science, 2004. **14**(03): p. 329-366.
138. Mostafa, M. and B. Bonakdarpour. *Decentralized runtime verification of LTL specifications in distributed systems*. in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. 2015: IEEE.
139. Chauhan, H., et al. *A distributed abstraction algorithm for online predicate detection*. in *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*. 2013: IEEE.
140. Flanagan, C. and P. Godefroid, *Dynamic partial-order reduction for model checking software.* SIGPLAN Not., 2005. **40**(1): p. 110-121.
141. Pervez, S., et al., *Practical Model-Checking Method for Verifying Correctness of MPI Programs*, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007. Proceedings*, F. Cappello, T. Herault, and J. Dongarra, Editors. 2007, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 344-353.
142. Siegel, S.F., *Model Checking Nonblocking MPI Programs*, in *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007. Proceedings*, B. Cook and A. Podelski, Editors. 2007, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 44-58.
143. Hilbrich, T., et al., *Distributed wait state tracking for runtime MPI deadlock detection*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, ACM: Denver, Colorado. p. 1-12.
144. Vakkalanka, S., et al., *Implementing Efficient Dynamic Formal Verification Methods for MPI Programs*, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, September 7-10, 2008. Proceedings*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Editors. 2008, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 248-256.
145. Vo, A., S. Vakkalanka, and G. Gopalakrishnan, *ISP Tool Update: Scalable MPI Verification*, in *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, S.M. Müller, et al., Editors. 2010, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 175-184.
146. Gopalakrishnan, G.L. and R.M. Kirby. *Runtime verification methods for MPI*. in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 2008.
147. Fard, F.H. and B.H. Far. *A method for detecting agents that will not cause emergent behavior in agent based systems - A case study in agent based auction systems*. in

Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on. 2012.
148. Fard, F.H. and B.H. Far. *Detecting distributed software components that will not cause emergent behavior in asynchronous communication style*. in *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on*. 2013: IEEE.
149. Fard, F.H. and B.H. Far, *Detecting Neutral Nodes in a Network of Heterogeneous Agent Based System*, in *Social Networks: Analysis and Case Studies*. 2014, Springer. p. 41-60.
150. Fard, F.H. and B.H. Far. *Detection and verification of a new type of emergent behavior in multiagent systems*. in *Intelligent Engineering Systems (INES), 2013 IEEE 17th International Conference on*. 2013.
151. Mousavi, A. and B.H. Far, *Eliciting Scenarios from Scenarios*, in *SEKE*. 2008. p. 466-471.
152. Mitchell, B., *Resolving race conditions in asynchronous partial order scenarios.* IEEE Transactions on Software Engineering, 2005. **31**(9): p. 767-784.
153. Fard, F.H. and B.H. Far. *Detection of implied scenarios in multiagent systems with clustering agents' communications*. in *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*. 2014: IEEE.
154. Fard, F.H. and B.H. Far, *On the Usage of Network Visualization for Multiagent System Verification*, in *Online Social Media Analysis and Visualization*. 2014, Springer. p. 201-228.
155. DELOACH, S.A., M.F. WOOD, and C.H. SPARKMAN, *MULTIAGENT SYSTEMS ENGINEERING.* International Journal of Software Engineering and Knowledge Engineering, 2001. **11**(03): p. 231-258.
156. DeLoach, S.A., *The MaSE Methodology*, in *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, F. Bergenti, M.-P. Gleizes, and F. Zambonelli, Editors. 2004, Springer US: Boston, MA. p. 107-125.
157. Mousavi, A., et al., *Strong Safe Realizability of Message Sequence Chart Specifications, International Symposium on Fundamentals of Software Engineering*, F. Arbab and M. Sirjani, Editors. 2007, Springer Berlin / Heidelberg. p. 334-349.
158. Fard, F.H. and B.H. Far. *Detecting a certain kind of emergent behavior in multi agent systems applied on mase methodology*. in *Electrical and Computer Engineering (CCECE), 2013 26th Annual IEEE Canadian Conference on*. 2013: IEEE.
159. Macal, C.M. and M.J. North, *Agent-based modeling and simulation*, in *Winter Simulation Conference*. 2009, Winter Simulation Conference: Austin, Texas. p. 86-98.
160. Macal, C.M. and M.J. North, *Agent-based modeling and simulation: ABMS examples*, in *Proceedings of the 40th Conference on Winter Simulation*. 2008, Winter Simulation Conference: Miami, Florida. p. 101-112.
161. Piao, C., X. Han, and S. Zhang. *Multi-agent modeling and analysis for E-commerce transaction network based on CAS theory*. in *2009 IEEE International Conference on e-Business Engineering*. 2009: IEEE.
162. Muscholl, A. and D. Peled, *Deciding properties of message sequence charts*, in *Scenarios: Models, Transformations and Tools*. 2005, Springer. p. 43-65.
163. MOSHIRPOUR, M., A. MOUSAVI, and B.H. FAR, *DETECTING EMERGENT BEHAVIOR IN DISTRIBUTED SYSTEMS USING SCENARIO-BASED SPECIFICATIONS.* International Journal of Software Engineering and Knowledge Engineering, 2012. **22**(06): p. 729-746.

164. Moshirpour, M., A. Mousavi, and B.H. Far. *Model based detection of implied scenarios in multi agent systems*. in *Information Reuse and Integration (IRI), 2010 IEEE International Conference on*. 2010.
165. Moshirpour, M., P. Mohassel, and B.H. Far. *Model based analysis of internet filtering systems*. in *Information Reuse and Integration (IRI), 2011 IEEE International Conference on*. 2011.
166. Fard, F.H. and B.H. Far, *Clustering Social Networks to Remove Neutral Nodes*, in *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*. 2012, IEEE Computer Society. p. 1289-1294.
167. Adhau, S.R. and M. Mittal, *A multi-agent based approach for dynamic multi-project scheduling.* International Journal of Advanced Operations Management ICOREM, 2011. **3**(3-4): p. 230-238.

Appendix: Permission of copy right material

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of Calgary's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to
http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

Title:	A method for detecting agents that will not cause emergent behavior in agent based systems - A case study in agent based auction systems -
Conference Proceedings:	Information Reuse and Integration (IRI), 2012 IEEE 13th International Conference on
Author:	Fatemeh Hendijani Fard; Behrouz H. Far
Publisher:	IEEE
Date:	8-10 Aug. 2012

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Title:	Detecting distributed software components that will not cause emergent behavior in asynchronous communication style
Conference Proceedings:	Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on
Author:	Fatemeh Hendijani Fard; Behrouz H. Far
Publisher:	IEEE
Date:	14-16 Aug. 2013

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Title:	Detection of implied scenarios in multiagent systems with clustering agents' communications
Conference Proceedings:	Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on
Author:	Fatemeh Hendijani Fard; Behrouz H. Far
Publisher:	IEEE
Date:	13-15 Aug. 2014

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

Title:	Detecting a certain kind of emergent behavior in multi agent systems applied on mase methodology
Conference Proceedings:	Electrical and Computer Engineering (CCECE), 2013 26th Annual IEEE Canadian Conference on
Author:	Fatemeh Hendijani Fard; Behrouz H. Far
Publisher:	IEEE
Date:	5-8 May 2013

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.