UNIVERSITY OF CALGARY

Inference of Emergent Behaviours of Scenario-Based Specifications

by

Abdolmajid Mousavi

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

April, 2009

© Abdolmajid Mousavi 2009

Abstract

This research provides a set of methodologies for detecting hidden behaviours of scenario-based specifications based on Message Sequence Charts (MSCs).

The main issue that the thesis is centered around is the emergent behaviours that can arise when the behaviour model of a system with multiple components is synthesized from its scenario-based specification. These unexpected behaviours can arise due to two separate effects. On one hand, due to the scenario-based specification and regardless of how behaviour models are developed, the synthesized behaviour model might introduce unexpected behaviours for the system. These unexpected behaviours are called emergent scenarios. On the other hand, depending on the assumptions and the technique used in the synthesis process, emergent behaviours might arise in the behaviour models of the components.

In detecting emergent scenarios, we deal with a problem where any component-wise distributed implementation of a system presents such behaviours for the system that do not appear in its scenario-based specification while the behaviours of components are in accordance to the specification. The thesis has some contributions in this part. First, the concept of stuck states is introduced, which not only covers the system deadlocks, but also addresses deadlocks for the system components. As a result, a new notion of realizability called strong safe realizability will be introduced for scenario-based specifications, which captures more hidden behaviours for them than the previously reported realizability notions. Second, we have developed an algorithm for checking realizability problems of MSC specifications in asynchronous setting. Third, it will be shown that two seemingly separate phenomena, namely, emergent scenarios as a system level artifact and emergent behaviours for the system components are both caused by the same local property developed in the behaviour of system's components.

While the main tendency in the thesis is to detect emergent behaviours, a new approach will also be presented for synthesizing behaviour models from scenarios whose advantage is in reducing the intervention of the domain expert in the synthesis process and also in producing consistent outputs when it is applied by different domain experts to the same application domain.

Acknowledgements

Firstly, I would like to express my fullest gratitude and appreciation to my supervisor Dr. Behrouz H. Far for his guidance and encouragement. Surely, without his technical knowledge and the freedom that he provided for me to do the work, I could not have reached at this point.

I am indebted to Dr. Jorg Denzinger for his critical and challenging discussions, which together with his useful feedbacks and stimulating suggestions helped me in improving my work. I am also obliged to Dr. Armin Eberlein who helped me in taking the first steps toward my Ph.D, and for his supprort throughout the program.

I also owe parts of the progress in my thesis to Dr. Sebastian Uchitel from Imperial College of London who kindly answered my inquiries regarding his PhD work as well as providing me with the LTSA-MSC tool and its related examples.

Finally, I am grateful to my family for all the support that they provided and for their patience throughout these years.

Table of Contents

Abstract . . . ii
Acknowledgements . . . iv
Table of Contents . . . v
List of Tables . . . vii
List of Figures . . . viii
1 INTRODUCTION . . . 1
1.1 Motiviation . . . 1
1.2 Scenario Language . . . 4
1.3 Behaviour Models . . . 4
1.4 Emergent Scenarios . . . 5
1.5 A General Overview of Work . . . 7
1.6 Application in Software Engineering . . . 9
1.6.1 The set of Scenarios M . . . 10
1.6.2 The Domain Theory . . . 10
1.6.3 Construction . . . 11
1.6.4 Realization . . . 13
1.6.5 Validation . . . 14
1.7 Contributions . . . 14
1.8 Thesis Outline . . . 15
2 RELATED WORK . . . 17
2.1 Scenarios . . . 17
2.2 Sequence Charts . . . 20
2.2.1 Basics . . . 20
2.2.2 Managing Multiple Sequence Charts . . . 25
2.2.3 Semantics . . . 31
2.2.4 Analysis . . . 33

I EMERGENT BEHAVIOURS FOR THE COMPONENTS OF SCENARIO-BASED SPECIFICATIONS 37
3 FROM SCENARIOS TO STATE MACHINES: INFERRENCE OF IDENTICAL STATES . . . 38
3.1 Background . . . 39
3.2 Definitions . . . 40
3.3 The Domain Knowledge . . . 45
3.3.1 Domain Theory . . . 45
3.3.2 State Value . . . 47
3.4 Behaviour Modeling . . . 49
3.4.1 Identical States . . . 50
3.4.2 Capturing The Domain Knowledge . . . 51
3.4.3 Domain Theory for the ATM Example . . . 53

3.5 Summary 54
4 FROM SCENARIOS TO STATE MACHINES: EMERGENT BEHAVIOURS 57
4.1 Background 57
4.2 Generalization 60
4.2.1 Assumptions 60
4.2.2 Criteria for Merging Identical States 60
4.3 Synthesis Algorithm 66
4.4 Emergent Behaviours 66
4.5 Harnessing Overgeneralization 67
4.6 Summary 69

II EMERGENT SCENARIOS FOR SCENARIO-BASED SPECIFICATIONS 73
5 SAFE REALIZABILITY AND IMPLIED SCENARIOS 74
5.1 Background 75
5.2 Definitions 78
5.3 Implied pMSCs and Indeterministic Behaviour of Processes 80
5.4 MSC Specifications 85
5.4.1 Some Choice Node Effects 85
5.4.2 Safe Realizability for MSC Specifications 89
5.4.3 Basic Executions 95
5.5 Algorithm 108
5.5.1 Complexity of the Algorithm 109
5.5.2 The Bottleneck of Complexity 111
5.6 Summary 112
6 STRONG SAFE REALIZABILITY AND EMERGENT SCENARIOS . . 114
6.1 Introduction 114
6.2 Background 116
6.3 Yet Another Non-Determinism 117
6.3.1 Strong Safe Realizability 118
6.4 Summary 122
7 APPLICATION EXAMPLE 124
7.1 Detecting Emergent Scenarios 124
7.2 Correcting MSC Specifications 127
8 SUMMARY AND OUTLOOK 130
8.1 Summary of Contributions 131
8.2 Outlook 131
A MSC ANALYSIS TOOLS 134
Bibliography 141

List of Tables

3.1 A part of the domain theory for ATM ($T_{ATM}(q_3^{m2})$). 54
3.2 Another part of the domain theory for ATM ($T_{ATM}(q_7^{m2})$). 54

List of Figures

1.1 A framework for scenario-based software development in which the results of the thesis can be applied. 9

2.1 Example for a scenario using narrative text. 18
2.2 Example of a sequence chart. 20
2.3 Elements of a sequence chart. 21
2.4 Sequence chart with event labels. 23
2.5 A high-level Message Sequence Chart. 27
2.6 Sequence chart for an unreadable swipe card scenario. 27
2.7 Component and system states in sequence charts. 30
2.8 Expressing alternative behaviour using scenario composition. 30

3.1 If queues are non-FIFO, there is no way for process C2 to know whether it is receiving the first or the second message x. 42
3.2 Non-FIFO condition allows for crossing of dissimilar messages. 42
3.3 A preliminary set of scenarios for an ATM system. 44
3.4 eFSM for the ATM process in m2. 44
3.5 A case where state values can distinguish states of process $C1$ in a) and b), while global system variables cannot 49
3.6 The result of merging identical states for the ATM example. 54

4.1 A framework for using emergent behaviours. 59
4.2 A preliminary set of scenarios for an ATM system. 59
4.3 A state machine for ATM extracted from m_2. 60
4.4 Two identical states q_s and q_t of state machines A and B are merged. . . 62
4.5 Flowchart for synthesizing a behaviour model for process i from scenarios. 70
4.6 a) A state machine obtained from Figure 4.3 by merging states $q_3^{m_2}$ and $q_7^{m_2}$; b) An emergent behaviour for ATM that can be added to the set of scenarios of Figure 4.2. 71
4.7 Sequence diagram for an Alarm Clock system. 71
4.8 State machine for the controller process. 71
4.9 Behaviour model of the controller process when identical states are merged without checking the criteria of Definition 8. 72

5.1 A process control system. 76
5.2 Two scenarios (MSC1 and MSC2) for the system of Figure 5.1. These scenarios can imply another scenario (MSC3) that could be unsafe for the system. 76
5.3 A more realistic version of MSC1, containing measurements messages. . . 77
5.4 Two MSCs. 81
5.5 Two state machines for describing behaviour of process C1 in two scenarios of Figure 5.4. 81

5.6 The union of two state machines of Figure 5.5. 81
5.7 An implied pMSC obtained from MSC4. 81
5.8 Two MSCs. 82
5.9 An implied pMSC obtained from MSC6. 82
5.10 MSCs for illustrating choice nodes. 88

6.1 According to these two scenarios, process C1 can indeterministically decide to send or not to send message e. 118
6.2 An emergent scenario from two scenarios of Fig. 6.1. 118

7.1 MSC specification for a Boiler Control system. 126
7.2 LTS models of the processes in the Boiler Control system. 127
7.3 Two MSCs obtained from two basic executions of the hMSC of Figure 7.1. 127
7.4 Two emergent pMSCs obtained from two MSCs of Figure 7.3. 128
7.5 Modified Register scenario. 129
7.6 Corresponding MSCs of the basic executions s_1 and s_2 after modifying the Register MSC. 129

Chapter 1

INTRODUCTION

1.1 Motiviation

One frequent problem that requirements engineers are faced during system development is that stakeholders may have difficulties in expressing goals, properties, and requirements in abstract [1, 2]. This is particularly a problem in software development where uncertainties about software goals, functionalities, and properties increase significantly due to a wide range of different factors such as human interaction, compatible platforms and hardware, market competition, and so on.

One of the effective approaches for software and system development is to let the stakeholders start with describing system dynamics using scenarios. Typical scenarios of using the hypothetical system are sometimes easier to be grasped in the first place than some goals that can be made explicit only after deeper understanding of the system has been gained.

A scenario is a temporal sequence of interaction events among different processes, agents, or components. Scenarios describe how system components, the environment and users work concurrently and interact in order to provide system level functionality. Their simplicity and intuitive graphical representation facilitates stakeholder involvement and makes them popular for conveying and documenting requirements. Each scenario is only a partial description of system requirements, which, when combined with all other scenarios contributes to the overall system description.

Scenarios can be used for a wide variety of purposes in the requirements engineering life cycle such as eliciting requirements and generating acceptance test cases. They also

can be used easily by multiple stakeholders with different background to build a shared view. However, while scenarios are widely used, their use is mostly without a precise and formal semantics [3, 4, 5]. Consequently, tools such as Rational Rose ([6]) are often used for syntactic consistency checks. Nevertheless, recently there also has been work on providing precise semantics to scenario-based specifications([7, 8, 9, 10]) and developing semantic analysis tools [11, 12, 13].

Although very expressive, scenario-based specifications are prone to subtle errors centered around two main drawbacks with respect to analysis and validation i.e. incompleteness and partial description. In order to mitigate these problems, reliable development techniques and support for system behaviour analysis seems to be essential for scenario-based software development. This is particularly true for distributed systems comprising of multiple autonomous components or processes such as PCs, ATM machines, network applications, operating systems, and more importantly where subtle errors in the specification might cause life threatening errors in the final product [14]. Consequently, complementary techniques such as behaviour modeling as well as syntax and semantics analysis tools that mitigate those weaknesses brings more reliability to the scenario-based development paradigm. Specifically, our main concerns for the work in this thesis are the following.

1. Similar to test cases, scenarios are also partial descriptions of system's behaviour. This results in completeness and coverage problem, which makes it difficult to verify the absence of errors such as incompleteness with respect to the system requirements [15] or conflicts among requirements[16].

2. Scenarios are only instances of system's behaviour and so, they need to be properly combined in order to have a full description of system's behaviour.

3. As safety and liveness properties are implicit in a program trace, scenarios are

not an expressive language for the required properties about the intended system. These properties finally need to be made explicit in order to support analysis and implementation.

Issue 1 implies that it is not realistic if we assume that in the first draft stakeholders come up with a set of scenarios that cover all possible system behaviours. In fact, existing analysis techniques developed under the assumption of complete (or even relatively complete) specifications cannot tackle the partial nature of scenario-based specifications. In particular, overlooked functionalities and emergent behaviours are inherent to the partial nature of scenarios, which if not detected, can result in unexpected behaviours and costly errors later on in the run time. In this regard, this thesis includes methodologies for detecting emergent behaviours resulted from scenario-based specifications.

Issues 2 and 3 are respectively concerned with the instance-based nature of scenarios and that high-level requirements are not explicitly represented. For instance, for an ATM a requirement such as "every card inserted shall be returned unless the password entered remains invalid after three attempts" cannot be explicitly expressed using system traces expressed in scenarios. In this sense, scenarios will not be requirements, even though they frequently appear in preliminary material for requirements elicitation - much the same way as examples of input-output pairs are not specifications of a program and program traces do not describe an algorithm. To address issue 2, this thesis includes a new approach for synthesizing system's behaviour models from instance scenarios. The approach benefits from generalizing instance behaviours in scenarios and timing abstraction. In addition to addressing issue 2, generalization and timing abstraction help in mitigating issue 3 because they serve as an inference mechanism by which high level requirements and system properties will be highlighted in behaviour models.

1.2 Scenario Language

One of the most widespread approaches for documenting scenarios is using Message Sequence Charts (MSCs) [8] (or UML sequence diagrams [17]). In sequence charts, vertical lines called components, instances, or processes are used to describe the entities that participate in the scenario (see Figure 2.2 in Chapter 2). Components can be used to represent various kinds of entities such as actors, agents or objects. Interactions between components are shown by horizontal arrows called messages. The source of a message indicates which component initiates the interaction. Messages labels are normally considered to define the type of interaction, thus having the same sending and receiving components throughout all scenarios. The points on components where an arrow starts from and finishes are called send and receive events. An event can be considered the phenomenon observed by a component because of an interaction. A sequence chart is interpreted time-wise in a top-down fashion; an event on a component occurs before all other events of the same component that appear below it. In addition, the direction of messages also provides information on the order in which events occur; a receive-event must never occur before its corresponding send- event. For this reason, as a convention, messages are required to be drawn horizontally or with a downward slope. These assumptions on how time is represented in sequence charts determine a partial ordering of events [8].

1.3 Behaviour Models

A scenario language such as MSC notation describes both system executions and system components. It describes the system components that are involved in providing the intended system behaviour and the messages they are capable of sending and receiving. However, each scenario is like a window where only a part of interaction between system

components is visible. Consequently, in order to have the behaviour of each component separately from other components, we need a mechanism to represent the behaviour of each component as a single entity. This is where behaviour models help by providing a single model for each component. The joint behaviour of all components in terms of concurrent execution of their behaviour models represents the system model.

State machines are one of the popular behaviour modeling languages that is also adopted by the UML framework for software development [17]. State machines with various syntax, semantics, and extensions such as Statecharts [18], and RSML [19], have been extensively used for rigorous analysis and mechanical verification of systems properties. However, in this thesis we restrict ourselves to basic state machines defined by a set of states, initial and final states, and the transition relation [20]. We also use Labeled Transitions Systems (LTSs) to model systems with infinite executions. For our purpose, state machines and LTSs are similar except that having accepting states in LTSs is optional. This relaxation in having accepting states makes it possible to have infinite runs in LTSs in order to model reactive systems with infinite executions. We use a state machine or LTS to model each component and use transitions labels to model the messages components send or receive. We also use concurrent execution of multiple state machines (LTSs) as the system model and call it a distributed implementation for the system.

Chapter 3 presents our approach for synthesizing component's behaviour models from scenario-based specifications.

1.4 Emergent Scenarios

The term implied scenarios is initially coined in [21] to name any behaviour that is executed by every distributed implementation of an MSC specification while that behaviour is not explicitly specified in the specification. However, for two reasons, we decided to use

emergent behaviours instead of implied scenarios. First, as it will be shown in Chapter 6, due to our extended definition of deadlocks for scenario specifications, a new class of unspecified behaviours might happen for MSC specifications where the underlaying definition of implied scenarios does not cover them. Second, in this thesis we will be dealing with both the system and the component behaviours. As a result, possible new behaviours for the components that are not specified by scenarios would be important as they can provide useful feedbacks for correcting the specifications. This unspecified behaviours for components are not also covered by the definition of implied scenarios.

Two types of emergent behaviours will be studied in this thesis: i) emergent behaviours for components; ii) emergent behaviours for the system, which are also called emergent scenarios. In the component level, emergent behaviours happen because of the generalization mechanism by which state machine models are inferred from example scenarios. These extra behaviours are not inherent to the specification and depend solely on the assumptions and the generalization technique used in the synthesis approach. This is the reason that they have been referred to in the literature as a side effect of generalization called overgeneralization [22]. It should be noted that emergent behaviours for components are not necessary unwanted behaviours. Sometimes they may be just considered as unexpected situations due to specification incompleteness.

In Chapter 4, we present a set of criteria for detecting emergent behaviours for components.

On the other hand, system level emergent behaviours (or emergent scenarios) are inherent to the specification and regardless of the synthesis approach, they exist in any distributed implementation of the specification. Such emergent scenarios arise because the intended behaviours in different scenarios can combine in unexpected ways when each component has only its own local view in the scenarios. Analogous to emergent behaviours for components, emergent scenarios are not always considered unwanted be-

haviours. Instead, they might represent an underspecification that need to be detected and fixed.

Chapters 5 and 6 present our approach for detecting emergent scenarios. While some works (e.g. [23]) detect emergent scenarios (in fact, all previous works only detect implied scenarios because they assume a different definition of deadlocks for scenario-based specifications) by building and then comparing behaviour models with the specification, one of the advantages of our work is in the use of a set of criteria defined directly on scenarios for detecting emergent scenarios. By so doing, we can refrain from the burden and the complexity of building system behaviour models. Furthermore, by checking the criteria on the specification, our approach is more effective when it comes to track the cause of emergent scenarios back to the specification and effectively correct the problem.

A closely related concept to emergent scenarios that will be addressed in Chapters 5 and 6 is realizability of MSC specifications [21, 24, 25, 26, 27]. A realization or a distributed implementation of an MSC specification for a system with multiple components is defined as the concurrent execution of the behaviour models of the components. If there exists such a realization whatsoever that is deadlock free and shows exactly the behaviours specified by the specification, it is said that the specification is safely realizable. If on the other hand, there is no such a realization, there would be some emergent scenarios that are not part of the specification but part of the behaviour of any distributed implementation covering the specification. Thus, safe realizability and emergent scenarios are so defined that existence of one is an indication for the absence of the other.

1.5 A General Overview of Work

The main objective of this research is to develop methodologies for ameliorating some of the shortcomings of scenario-based specifications. The thesis is presented in two main

parts. Part I includes Chapters 3 and 4 and studies scenario-based specifications in the component level where a set of techniques will be presented for inferring emergent behaviours for the components of a system from its scenarios-based specification. This part includes a domain theory for the purpose of synthesizing behaviour models from scenarios given in MSC notation; a new approach for synthesizing behaviour models from scenarios; and a set of criteria to be checked for emergent behaviours of the components. Part II includes Chapters 5, 6, and 7 and studies scenario-based specifications in the system level where with no emergent behaviour allowed for the components, scenario-based specification are analyzed for detecting possible emergent scenarios for the system. This part involves a new notion of realizability for MSC specifications and an algorithm for detecting emergent scenarios.

The benefits of our work can be better envisaged in a traditional software development framework. Figure 1.1 sketches such a framework in which the results of this thesis can be applied. The gray parts show the contributions of the thesis. The framework begins with synthesizing behaviour models of components from a preliminary set of scenarios given as the initial approximation of the system behaviour. This is indicated by lines 1 to 5 in the figure. Then, the result of behaviour modeling (e.g. feedbacks from emergent behaviours for components) can be used to correct the set of scenarios (lines 6 to 8).

Also, we can have an analysis phase in the system level in which any mismatch between the specification and its component-wise distributed implementation will be found and possible emergent scenarios are detected (lines 10 to 13). After being detected, these emergent scenarios also provide another feedback for correcting the set of scenarios (line 8). The process of behaviour modeling and analysis can continue until a satisfiable specification is obtained.

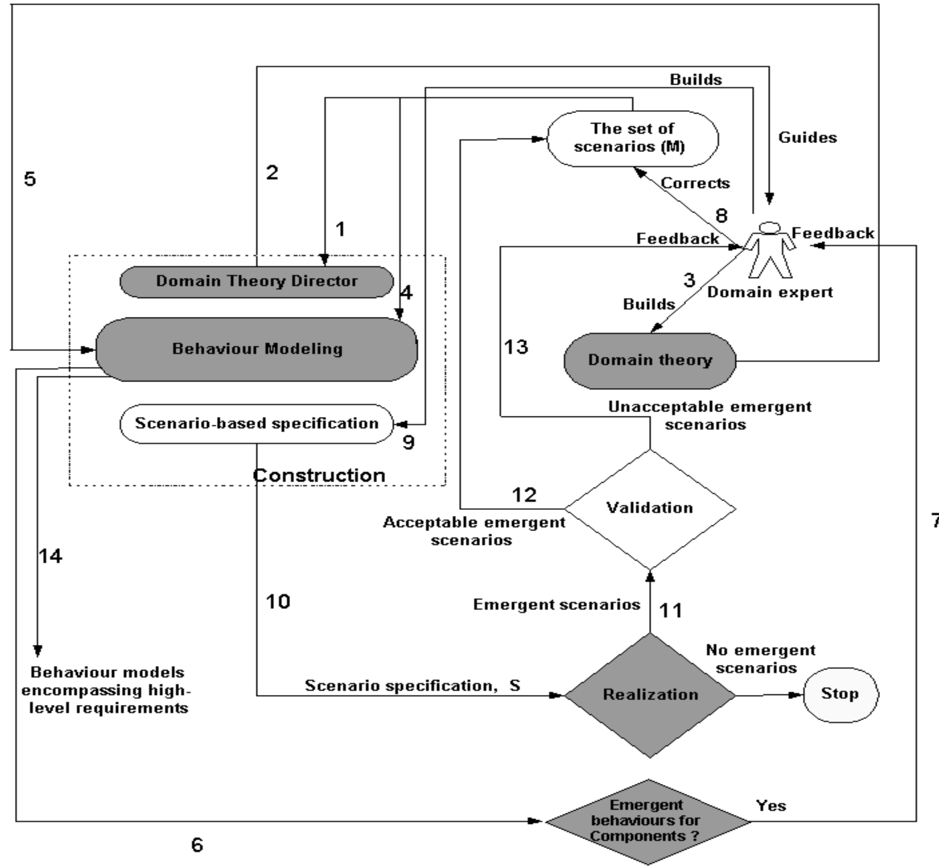


Figure 1.1: A framework for scenario-based software development in which the results of the thesis can be applied.

1.6 Application in Software Engineering

Let's demonstrate the contributions of the thesis using Figure 1.1. We start with the construction in the figure where for each component i of system, a **domain theory** D_i is built from a preliminary set of scenarios M (lines 1, 2, and 3). A **domain theory director** guides the domain expert to effectively build the domain theory. The domain theory D_i and the set of scenarios M provide the required inputs for **behaviour modeling** (lines 4 and 5). Then, component's behaviour in the behaviour model can be analyzed for any deviation (emergent behaviours) from the scenario set M (line 6), and then, the set of scenarios M will be modified based on the result of this comparison (lines 7 and 8). Furthermore, at the same time a scenario-based specification S (explained in Sections

1.6.1 and 1.6.3) will be constructed by the domain expert (line 9).

Then, the specification S can be checked for **realizability** (line 10) to see whether there exists a distributed implementation that shows exactly the behaviours specified in S. If there exists such a realization, we can stop and S could be the final specification. If on the other hand, there is no such a realization, there would be some emergent scenarios that are not part of S but part of all implementations of S. These emergent scenarios are validated against system's properties and goals to provide the required feedback for correcting the specification (lines 11, 12, and 13).

The construction-realization-validation process can be applied to all components of the system until either there is no change in the specification when we switch from one component to another or the domain expert decides to stop the process.

1.6.1 The set of Scenarios M

The first approximation of system specification is given by a preliminary set of scenarios M (which for our purpose, it is given as Message Sequence Charts (MSCs). Later, by the feedbacks received from the construction and the realization phases, M will be corrected by the domain expert and a scenario-based specification will be constructed.

We distinguish between the set of scenarios M and the scenario-based specification S, since the latter can include high-level Message Sequence Charts - a graph that allows for the combination of MSCs in sequential, alternative, and iteration forms [8] (high-level Message Sequence Charts are formally defined in Chapter 5).

1.6.2 The Domain Theory

The domain theory is a set that its members are selected pairs of messages that a component sends or receives in M. These pairs of messages have a special relation to each other called **semantical causality** (see Chapter 3). A message b is a semantical cause

for message c, if the component keeps the result of the operation of message b in order to perform message c. As an example, for an ATM, the *insert card* is a semantical cause for the *eject card* because the ATM must keep the card inserted in order to eject the card.

Since semantical causality captures dependency of one message on other messages, it will be an invariant property for a system. Having the domain theory based on system's invariants makes it reusable and consistent among different domain experts since system's invariants are inherent system properties and independent of the system expert.

Contribution of the thesis: *A domain theory in the form of a set of selected pairs of messages with semantical causality relation (Chapter 3).*

1.6.3 Construction

The construction is where behaviour models and scenario-based specifications are constructed from the set of scenarios M. It consists of the **domain theory director** and the **behaviour modeling**.

Domain Theory Director

The purpose of the domain theory director (see Chapter 3) is to make the construction of the domain theory easier for the domain expert. For a given component, the domain theory director guides the domain expert to focus on selected pairs of messages from all the messages in M and find the semantical causality relation between them. These pairs of messages are selected based on the requirements of behaviour modeling.

Contribution of the thesis: *A method that from a pool of messages exchanged between system's components in scenarios, selects only pairs that are effective for behaviour modeling and queries the domain expert about the semantical causality relation between them (Chapter 3).*

Behaviour Modeling

Generally, synthesis of behaviour models from scenarios needs a proper generalization technique while avoiding overgeneralization. Like related works [22, 28, 29], for generalization we detect identical states of components in different scenarios (though with our own technique that would be explained in Chapter 3). However, unlike other approaches, after detection, identical states are conditionally merged in order to avoid overgeneralization.

The condition for merging identical states will be defined in Chapter 4, which is a local property developed by the behaviour of system's components. Then, in Chapter 5 we will show that this is the same local property that explains realizability problems for scenario-based specifications. This result is notable in that it bridges between Part I and Part II of the thesis as two research areas that so far have been studied separately i.e. the problem of emergent behaviours for the system components (Part I) and the realizability problem that studies implied scenarios in the system level (Part II).

For identical states detection, based on the domain theory that is already constructed from the set of scenarios M, we use some of the messages that a component has already sent or received in a scenario to assign a state value to the current state of the component. Then, identical states of the component in different scenarios will be defined as the states with the same state values. The exact definition of state value will be given in Chapter 3. However, roughly speaking, for a given scenario the state value of a component is the ordered sequence of those messages that have occurred before its current state and are semantical causes of some messages that will happen after its current state.

Having states labeled with state values resembles guarded transitions in state machine requirements specification languages [19]. This fact can be utilized to infer declarative requirements [30] from scenarios in a state machine representation, which is shown in Fig. 1.1 as behaviour models encompassing high-level requirements (line 14). While in

the current practice, synthesizing behaviour models from scenarios results in unguarded transitions in the state machines, our behaviour modeling technique is able to abstract the timing between messages away and infer guarded transitions in the output state machines. The difference between state machines with unguarded transitions with those that have guarded transitions is that while the former can only show instances of the system behaviour, the latter also encompasses system's requirements in terms of **if** [condition] **then** [action].

Contributions of the thesis: *A new approach for synthesizing state machine behaviour models from scenarios (see Chapters 3 and 4).*

Scenario-Based Specification

This is where the set of scenarios M (represented by Message Sequence Charts) has undergone the necessary changes and corrections and possibly structured into a high-level Message Sequence Chart (see Chapter 5). HMSCs break the preliminary scenarios into smaller and manageable parts such that their sequential, alternative, and iterative composition covers the system behaviour. While it is not covered by our current work, one direction for future work is to use the result of behaviour modeling (since behaviour models are derived from MSCs and are augmented with the domain knowledge) for developing automated techniques for refactoring MSCs into hMSCs.

1.6.4 Realization

After the scenario-based specification S will be ready, it can be checked to see whether or not it is realizable. Specifically, strong safe realizability can be checked for the specification. Strong safe realizability is a new realizability concept introduced in this thesis in Chapter 6, which when holds, it implies safe realizability (the previous concept of realizability that is initially coined in [21]). Succinctly, strong safe realizability checks for possible emergent scenarios that are not explicitly specified in the specification but

can be inferred from it (see Chapters 5 and 6).

Possible emergent scenarios that are detected as the result of realizability analysis are used to enrich the original set of scenarios or otherwise provide the required feedback for correcting the specification. Although, this is not shown in Fig. 1.1, one of the advantages of our realizability analysis is in providing precise information regarding the location of the cause of emergent scenarios in terms of the scenario, the component, and the place in the communication history of the component where an emergent scenario is developed (see Chapter 7 for details).

Contributions of the thesis: *A new notion of realizability for MSC specifications called strong safe realizability, and an algorithm that given an MSC specification checks whether it is safely realizable or not (see Chapters 5 and 6).*

1.6.5 Validation

The validation phase can be performed by the domain expert who looks at an emergent scenario and decides whether to accept it and enrich the set of scenarios, or reject it and correct the set of scenarios such that the unwanted behaviour does not happen anymore. By localizing the cause of emergent scenarios in realizability analysis, the domain expert can specify the exact place in the scenario set where a correction is needed (see Chapter 7).

1.7 Contributions

This thesis is based on and extends several publications including [31, 32, 33, 34, 35, 36, 37], whose contributions can be summarized as:

- Domain knowledge is formalized in terms of a light domain theory that is based on invariant properties of software and systems. This contributes to the reusability

and the consistency of the domain theory (human factors are reduced)

- A new approach for synthesizing state machines behaviour models from scenarios
- A realizability model that extends safe realizability and captures more emergent behaviours and pitfalls for MSC specifications
- An algorithm is developed for checking safe realizability of MSC specifications in the presence of hMSCs and in asynchronous setting (we are not aware of another method for this problem)
- A question regarding the criteria under which MSC specifications can be implemented without deadlocks and emergent scenarios is answered
- Emergent scenarios (including implied scenarios) and overgeneralization (emergent behaviours of components) have been both localized in the behaviour of individual components

1.8 Thesis Outline

In this thesis, we present a set of methodologies for inferring emergent behaviours of scenario-based specifications. In Chapter 2, we discuss some background work on scenario-based specifications in general and sequence chart notations in particular. Part I of the thesis involves Chapters 3 and 4 and presents our method for inferring emergent behaviours of components through behaviour modeling. It consists of our approach for building the domain theory and inference of identical states (Chapter 3), as well as the criteria for merging identical states (Chapter 4).

Part II of the thesis involves Chapters 5, 6, and 7, and it is concerned with a problem different than Part I, that is, while no emergent behaviour is assumed for individual components of a system, we will see how emergent behaviours are possible for the system.

Chapter 5 discusses safe realizability and implied scenarios and also involves our method for verifying safe realizability and detecting implied scenarios. Moreover, the question regarding the criteria under which MSC specifications can be implemented without deadlocks and implied scenarios will be answered in this chapter.

Chapter 6 involves our extension of safe realizability and implied scenarios, namely, strong safe realizability and emergent scenarios. Also, Chapter 7 includes an intuitive example to illustrate the results of Chapter 5. Finally, Chapter 8 provides conclusions and possible extensions for the thesis.

Chapter 2

RELATED WORK

In this chapter, we give an account of the related work on scenario-based specifications in general and focus in more detail on issues that are related to sequence chart notations as these are extensively used in this thesis.

2.1 Scenarios

Scenarios are becoming an increasingly popular means for describing system behaviour. A scenario is a narrative description of how users, system components and the environment interact for the completion of a certain goal. Scenarios are stories, and as such, they are an effective means of facilitating communication among people. They are used in a variety of different settings that range from requirements engineering (e.g. [12]) and formal specifications (e.g. [10]) to code synthesis (e.g. [38]) and test case specification and generation (e.g. [39]). Within the literature, many different definitions of scenarios can be found (e.g. [3, 5, 40, 41]). There are significant differences in terms of the syntax, features, semantics and more significantly, in terms of their intent. Most of these differences are motivated by the setting in which scenarios are being described. However, there are some main aspects on which authors seem to coincide:

- Scenarios describe a sequence of events or activities [3].
- Scenarios refer to interactions between independent entities. Entities can be users, the system [3] (or possibly parts of it such as hardware, software, subsystems, objects [5]) and the environment [8].

The need to document scenarios has motivated the development of many scenario-based notations. Many authors have proposed using mixture of textual and pictorial techniques. These go from structured and narrative text [42], to storyboards of annotated cartoon panels, video recordings, and scripted prototypes [3, 43]. An example of a narrative text scenario is Figure 2.1. An important issue with scenarios is the level of abstraction

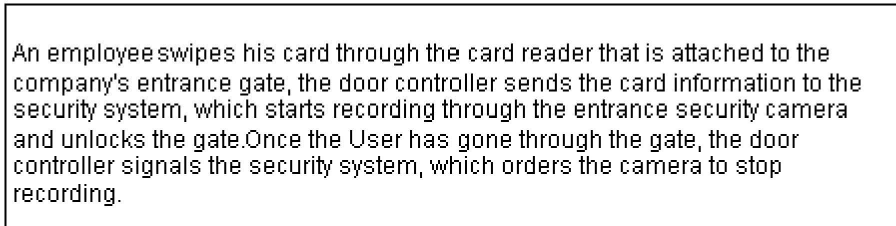

An employee swipes his card through the card reader that is attached to the company's entrance gate, the door controller sends the card information to the security system, which starts recording through the entrance security camera and unlocks the gate.Once the User has gone through the gate, the door controller signals the security system, which orders the camera to stop recording.

Figure 2.1: Example for a scenario using narrative text.

of their content. Scenarios can be concrete [3] or instance [44] scenarios that refer to specific entity names and argument values. Scenarios can also be more abstract, which are called type or abstract scenarios [45, 46]. However, we could make the scenario concrete by including for example the role of the employee, employee identifier, and security codes. The level of detail of the scenario can also be incremented by including more entities such as sensors or the network, or more interactions such as detailed aspects of the communication protocol between door controller and security system, or details of how the door controller detects that the user has gone through the gate. Furthermore, the scenario could be extended with alternative outcomes, for instance that the card is unreadable (as in scenario families in [44] or scenario trees in [47]). In addition, scenario pre- and post-conditions, such as that the scenario occurs in working hours and the status of the employee in the security database can be included (as in [48]).

Textual notations for documenting scenarios have been widespread in industry for some time [42]. However, these notations tend to be rather informal and, although useful for documentation and informal analysis, make automated processing of scenarios and

rigorous or semi-rigorous analysis very difficult. Thus, textual scenario specifications do not serve well our purpose of using rigorous and automated techniques for detecting emergent behaviours. There has been significant research on controlled English (e.g. [49]) and grammars for English (e.g. [50]), which constrain the structure of English sentences in order to make texts more amenable to automated processing. However, these techniques have yet to produce impact in industry in general. Furthermore, if controlled languages are to be used in requirements (or scenario) specifications the prospect is even more complex and further syntactic and semantics constraints for controlled languages must be introduced.

One of the most widespread approaches to documenting scenarios is using sequence charts. Sequence charts have been used to describe system behaviour for some time, and examples of their use can be traced back to the late 80's. They have been used, to exemplify service conventions in the International Standard Organisation's (ISO) Open Systems Interconnection (OSI) [51] and in several ISDN service recommendations of the International Telephone and Telegraph Consultative Committee (CCITT) [52, 53]. The latter, reconverted into the International Telecommunications Union (ITU), has undertaken a standardisation process resulting in a language called Message Sequence Charts (MSCs), which has undergone several revisions since its first version in 1992 [54, 55, 8, 56]. For an account on the standardisation of MSCs by the telecommunication industry, refer to [57].

In addition to being the starting point for MSCs, sequence charts constitute the core of a plethora of existing graphical scenario-based languages. The most famous one is the UML [17] notation for scenarios, sequence diagrams, which strongly resembles sequence charts. There are countless variations of sequence charts in the literature; in particular in research literature where a variety of different features and interpretations are being explored. As we show next, sequence charts have the advantage of being very intuitive

and simple, facilitating sequence chart elaboration and automated processing. For more on scenarios in general, refer to [3, 46].

2.2 Sequence Charts

In the remainder of this section we discuss sequence charts in order to introduce the main concepts related to scenario descriptions and address some of the existing syntactic and semantic variations that appear in other sequence chart-based languages. We present these variations by discussing different aspects of scenario-based specifications rather that describing each specific language separately. For a detailed survey on sequence charts refer to [58].

2.2.1 Basics

Sequence charts are a widespread graphical notation for documenting scenarios (see Figure 2.2 and Figure 2.3). Sequence charts consist of components, events, and messages which will be explained further down.

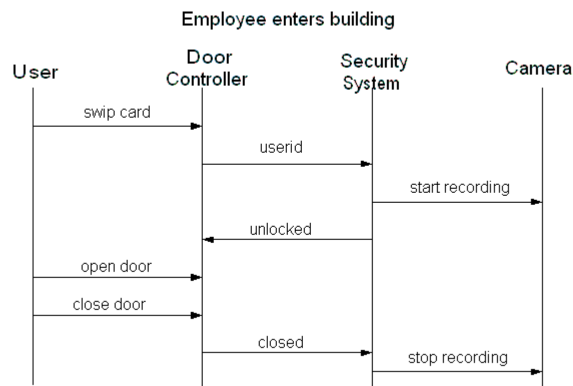


Figure 2.2: Example of a sequence chart.

Components

As mentioned in Chapter 1, in sequence charts vertical lines or components represent the entities participating in the scenario. By component, we mean a part or element

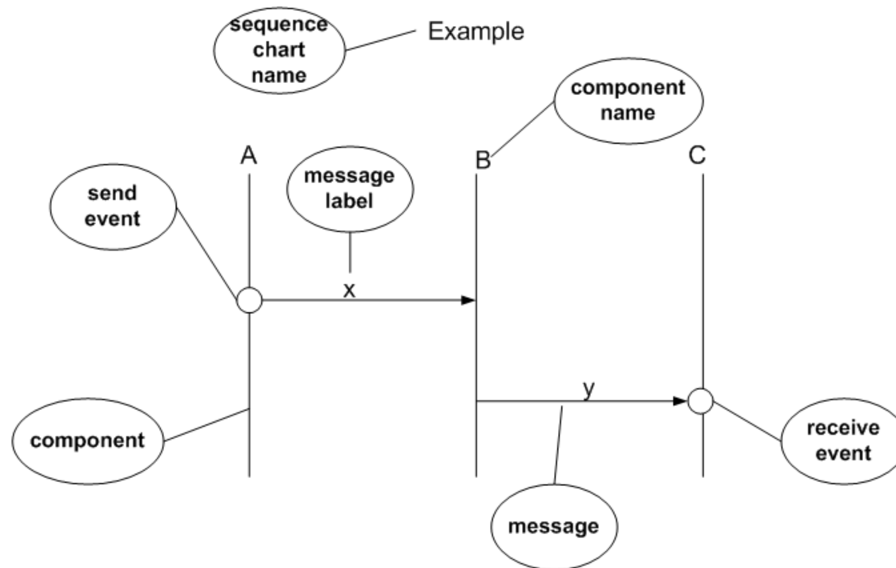


Figure 2.3: Elements of a sequence chart.

of a larger whole, which in our case is a system. In other words, by component we refer generically to software and hardware systems and subsystems, users, processes, and software components (as in [59] for example) among others. According to the context for which a particular scenario-based language has been designed, components can be given a more restricted interpretation. For example in a high-level approach to scenarios that focuses on the distinction between system and environment, scenarios might use one vertical line to represent the system to be developed and other lines only for external actors such as existing systems and people or organisations. On the other hand, in the context of UML, a lower level of abstraction can be used and components can model objects [40, 41]. Whatever the level of abstraction in which components are interpreted, the common assumption is that they represent independent entities that have capabilities for interacting with their environment.

Messages

Messages in sequence charts represent interactions between components. However, as

in the case of components, they are also given a range of interpretations according to the context in which sequence charts are used. In particular, the interpretation of a message relies heavily on the interpretation of the components that are involved in the interaction. For example, when at least one of the components involved is a person, messages model physical interactions. This includes examples such as typing data into a console, swiping a security card or delivering a package. In cases where people are not involved, a message can model a remote procedure call, message passing, method invocation, or an exception being raised and caught. Messages usually involve only two components, a sender and a receiver. However, messages can be used to model broadcast and multicast interactions.

Despite the different semantics that can be given to messages, three characteristics apply to all interpretations. Firstly, a message explicitly identifies the components involved in an interaction. Secondly, a message identifies, by means of the arrows source, the component that initiates the interaction. Thirdly, a message determines the point (an event) in which the interaction is observed by a component involved in the interaction.

Event Ordering and Synchronous/Asynchronous Messages

The semantics of messages impacts considerably on how time is modeled in a sequence diagram. As mentioned before, it is assumed that an event on a component occurs before all other events of the same component that appear below it. In addition, it is also assumed that a receive-event must never occur after its corresponding send-event. These two assumptions determine a partial ordering of events that specifies the relative order in which events may occur over time.

Let us return to the example of the employee entering a secure building, which we now depict in Figure 2.4 with labeled events only to facilitate the following explanation on event ordering. This scenario states, for example, that the reception of message stopRecording (event p) occurs after the sending of message startRecording (event k). In

other words that k < p. This is because event n is further down than k on the Security System component (k < n) and p is the receive-event corresponding to the send event n (n < p). However, the converse is not so, the reception of message stopRecording (event p) cannot occur before the sending of message startRecording (event k).

We say that the ordering of events is partial because not all pairs of events are temporally related. For example, there is no relative ordering between the reception of startRecording(event o) and the reception of message unlock (event f). This is reasonable if we think of messages as taking an unknown amount of time to reach their destination. Thus, different messages could have different delays, and thus the ordering of the reception of these messages cannot be assumed.

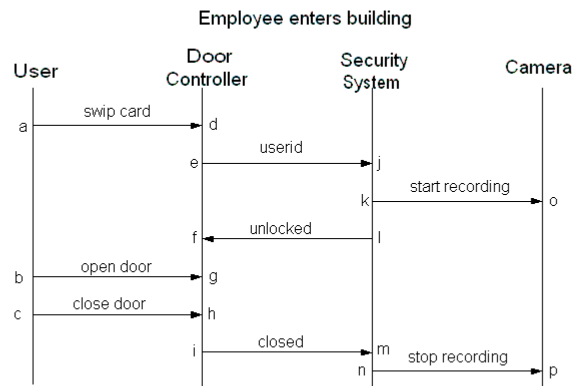


Figure 2.4: Sequence chart with event labels.

The partial ordering of events that we have presented is the common core for most approaches based on sequence charts and is based on minimal assumptions on the temporal relation between events. These assumptions can be too weak when using specific interpretations of messages. For example, consider the message swipeCard in Figure 2.4. The message corresponds to a physical interaction between a user that wishes to enter the company premises and a door controller placed at one of the entrances. In this case, the send-event (when the employee swipes the card) might be considered to occur at the same time as the receive event (when the card controller detects and-or reads the

swipe card). In other words, send and receive events can be considered to occur simultaneously. In these cases, the communication between components is called synchronous. In the case of message userId, where the card reader sends the card information to a remote database to validate access permissions of the employee, it may not be reasonable to consider synchronous communication. Network latency may need to be taken into account, forcing the receive-event (when the database receives the query) to occur after the send event (when the door controller sends the query). This kind of communication is called asynchronous. Existing scenario-based languages assume either synchronous or asynchronous communication or allow both. When considering both kinds of communication, different notations are used. For example, in sequence diagrams synchronous and asynchronous messages are differentiated by using filled and hollow arrowheads [60]. Sometimes, asynchronous messages are drawn with a downward slant to indicate that time elapses from its emission to its reception.

If we assume that messages starRecording and unlock model asynchronous communication, then the correct interpretation of the sequence chart is that the reception of the startRecording is temporally unrelated to that of unlock. Thus, message reception of both messages could occur in either order, not guaranteeing that the camera records when the employee goes through the door. If we assume that these messages model synchronous communication, then the correct interpretation of the scenario is that the security camera receives the order to start recording before the unlock message is even sent. The same situation holds if we assume that start and stop recording messages are asynchronous; when stopRecording is sent, it is not guaranteed that startRecording has been received. However, startRecording must occur before stopRecording.

In the context of asynchronous communication, additional constraints on the ordering of events can arise depending on the assumptions on the message queuing that handle the asynchronous messages [61, 62]. For example, if a unique first-in first-out (FIFO)

message queue is assumed to buffer all messages sent in the system, it would enforce a first-in first-out policy that would constrain certain event orderings. Such assumption would guarantee that unlock is received after startRecording is. However, it does not guarantee that unlock is sent after startRecording is received. Other assumptions that introduce different constraints on the ordering of messages are assuming each component has one FIFO queue that buffers the messages that are sent to it, or assuming that there is a FIFO queue for every pair of communicating components.

Extensions to Sequence Charts

There have been many extensions to the basic sequence chart notation presented above. We have already mentioned the inclusion of different message types to distinguish synchronous and asynchronous communication and queues to impose restrictions on event orderings. Other extensions include the explicit use of time to describe delays, timeouts and deadlines (e.g. [63, 8]), the dynamic creation and termination of components (e.g. [8]), parametric message and use of data (e.g. [64, 65]), local or component state labelling (e.g. [66]), global or system state labelling (e.g. [67]), liveness conditions [68], iterative and exceptional behaviour (e.g. [8]), and mechanisms for combining scenarios. We discuss the latter in more detail in following sections.

2.2.2 Managing Multiple Sequence Charts

Scenarios, and sequence charts in particular, are partial descriptions. One scenario conveys relatively little information as it describes one of usually infinite possible system behaviours. Thus, in the context of requirements engineering or system specification, scenarios need to be combined together to provide a more complete view of how a system is expected to behave. Typically, scenarios are provided by different stakeholders and address different system functionalities. The conjunction of all scenarios provides a

system description. However, choosing the right abstractions for combining scenarios is not a minor issue. Sequence charts have been extended in many very distinct ways in order to tackle this problem. Two of these ideas, namely scenario composition and state identification are used in this thesis and will be discussed here.

Scenario Composition

In the approach adopted by the International Telecommunication Union (ITU) [8], focus is on providing scenario specifications with a means for managing complexity. Simple sequences of behaviour are described using an extension of sequence charts called basic Message Sequence Charts (bMSCs). In addition, three basic constructs for combining bMSCs are provided: vertical, horizontal and alternative composition. Vertical composition of two bMSCs combines them sequentially. The system behaviour is determined by the behaviour of the first bMSC followed by the behaviour determined by the second one. Vertical composition introduces a subtle issue: events in the first scenario do not necessarily occur before all events of the second scenarios. The partial ordering of events determined by vertical composition is the result of the syntactical composition of the two scenarios: placing one bMSC at the bottom of another one and then linking the components they have in common [10]. Horizontal composition of two bMSCs amounts to considering they occur in parallel. In the case that both bMSCs have some or all components in common, it is assumed that the behaviour of the common component(s) is the interleaving of the behaviours of these component(s) in the separate bMSCs [10]. Alternative composition defines a set of possible MSCs that can be considered vertically composed with some initial MSC. In other words, it defines a set of alternative behaviours as the continuation of a given scenario. The underlying notion of scenario composition is that they can be used as building blocks to describe a behaviour that is more complex.

Several syntactic constructs, equivalent in terms of expressiveness [10], are provided by

the ITU standard for specifying scenario composition: inline expressions, MSC reference expressions and high-level MSCs, the last being the widely adopted (e.g. [61, 10, 67]). High-level Message Sequence Charts (hMSCs) are directed graphs where each node references either an hMSC or a bMSC. Edges indicate the acceptable ordering of scenarios, thus allowing stakeholders to reuse scenarios within a specification and to introduce sequences, loops, and alternatives of bMSCs. Figure 2.5 depicts an hMSC with two nodes, one of which refers to the sequence chart we have presented previously. In order for the hMSC to be well defined, the other node (Unreadable Swipe Card) should be referring to either existing bMSCs or hMSCs. An arrow with a black filled circle as its tail indicates the initial starting point of the hMSC. In addition, the smaller black circle is a convenient way of reducing the number of edges that the hMSC has, and in this way increasing understandability; it can be simply considered an empty scenario.

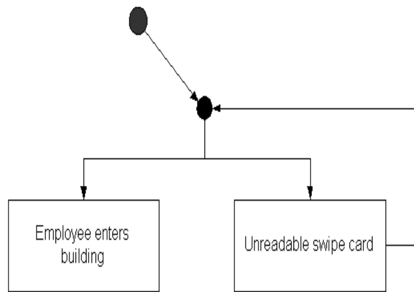


Figure 2.5: A high-level Message Sequence Chart.

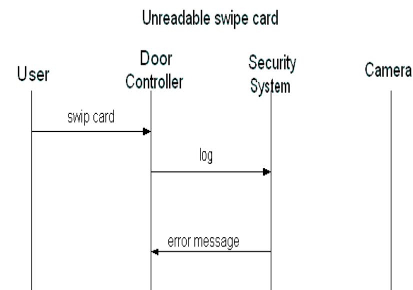


Figure 2.6: Sequence chart for an unreadable swipe card scenario.

The advantage of the hMSC approach is that it allows stakeholders to break up a scenario specification into manageable parts in a simple, intuitive, and operational way, and to show how these different parts relate. On the other hand, scenario composition can lead to a large number of very short scenarios that must be composed in complex ways to describe the system's overall behaviour. This mitigates against the advantage of using scenario notations for depicting significant portions of system behaviour. A common variation of hMSCs is that of high-level Message Sequence Graphs (e.g. [69]).

These correspond to flat hMSCs, in other words hMSCs in which nodes are only allowed to reference bMSCs.

State Identification

An approach that differs significantly from the scenario composition approach is the identification of common component or system states throughout a set of scenarios (e.g. [70, 66, 67, 71, 29]). The assumption here is that the scenario specification is describing a state-machine that models the behaviour of system components. Thus, components in a bMSC are considered to model both a set of states in the state-machine (which we call component states) and the events that fire state change (called labeled transitions). Thus, in the scenario specification, every space between consecutive events is called a scenario state (see Figure 2.7, right), and is considered to refer to component state. The relation between scenario and component states is many to one, meaning that several scenario states can refer to the same component state. Thus, scenario states in different bMSCs that refer to the same component state provide information of how the scenarios are related (see Figure 2.7).

There are two basic mechanisms for identifying component states. The first is to allow stakeholders to tag scenario states (e.g. [66]). Typically, labels that describe the state of the component are placed on scenario states (see Figure 2.7, left); if two states in a scenario appear with the same label, they are considered to refer to the same component state.

The second approach avoids the needs for explicit state labelling in scenarios, and instead provides rules for identifying component states. These rules are usually based on domain-specific knowledge and additional information of the system being specified. For example, SCED [70] synthesises statecharts [18] while applying some assumptions in order to decide whether two scenario states represent the same statechart state. Another

example is the work of Whittle and Schumann [29], which uses an Object Constraint Language (OCL) specification that states pre- and post-conditions for scenario messages. The OCL specification is traversed with the MSCs to produce a valuation of state variables for each scenario state. Scenario states that have equivalent valuations are considered to represent the same component states.

A variation of identifying component states is that of system states [67]. Instead of identifying component states on components, labels that cover all components of a scenario are used to mark a specific system state (see Figure 2.7), which essentially models the state in which each component is in at a particular moment. Identically labeled system states refer to the same system state.

An advantage of explicitly labelling component or system states is that they can be used to enrich scenario descriptions with additional information. This information can come from specifications that have different system viewpoints or from domain-specific knowledge. In addition, incorporating component or system state information may provide means for progressively moving into a more detailed design description. Compared with scenario composition, identifying states allows complex component behaviour including alternative behaviours to be described in bMSCs of any length. For example, in Figure 2.7, we have two bMSCs with a system state labeled Ready for operation. This means that if the system is in bMSC Example 1 and messages x and y occur, then, instead of message z occurring, the system could now continue with b and c. Thus, the system state has introduced alternative behaviour.

To introduce the same alternative using scenario composition, we would need to split the scenarios exactly where the "ready for operation" state is, to then introduce some composition mechanisms to establish all possible alternative behaviour (see Figure 2.8). Although the specified behaviour is the same, we have had to split scenarios into smaller parts (some with only one message in them!) that may not be meaningful on their own.

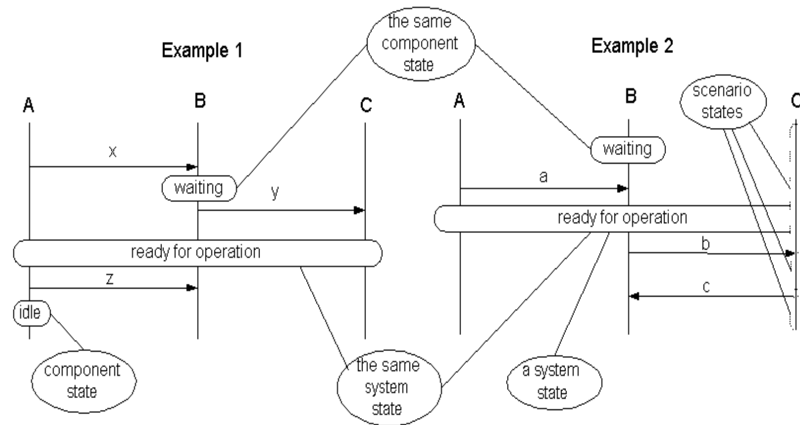


Figure 2.7: Component and system states in sequence charts.

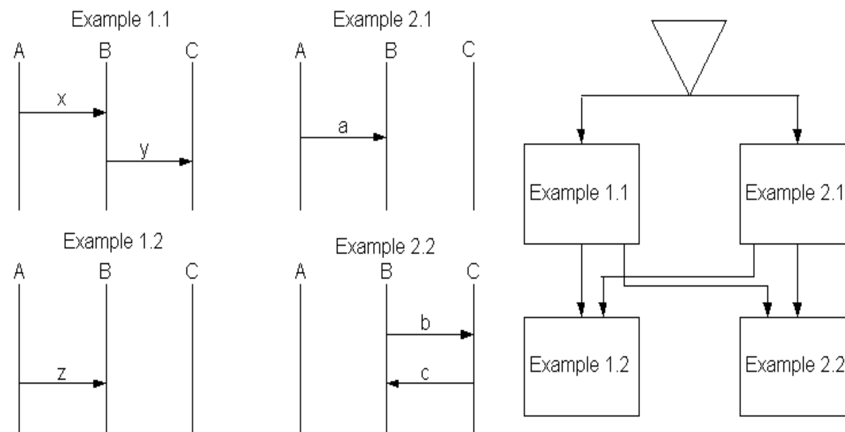


Figure 2.8: Expressing alternative behaviour using scenario composition.

The ITU provides a notation that resembles that of state labelling. Its MSC standard [8] includes the notion of local and global conditions, which could be interpreted as component and system states. However, ITU has not assigned any semantics to these syntactic elements. Several meanings to these constructs have been proposed. Component and system state identification is one of the possibilities being studied.

In UML [60], sequence diagrams allow introducing state information; however, it is not clear how this information affects the composition of different scenarios [4, 5, 40, 41]. To understand the relation between sequence diagrams it is usually necessary to refer to the statechart descriptions.

2.2.3 Semantics

In the previous section, we introduced the intuition of sequence charts and related concepts such as components, events, messages, interactions and time. We have also mentioned some of the different interpretations that exist for these concepts. We now discuss the techniques for defining these interpretations and more importantly the implications of such techniques on the purpose and utility of scenario-based specifications in the light of the development process as a whole.

Techniques

In terms of how semantics for scenario-based languages are defined, existing work is quite varied. We identify three broad categories: informal, algorithmic and abstract semantics. The first category corresponds to scenario languages with no precise semantics that are used in the context of informal development methods (e.g. [3]) and in some UML-based development methods such as the Unified Software Development Process [5] and others (e.g. [4, 40, 41]). Books presenting these methods usually devote one or two chapters to scenarios, providing general intuition in natural language supported with examples. Although useful for documentation and informal analysis, the lack of a precise interpretation of scenarios makes rigorous analysis of scenarios and formal verification of system compliance to requirements extremely difficult.

The second category includes approaches in which the semantics of a scenario specification is implicit, given by means of a translation algorithm. Using an algorithm to translate scenario specifications into other notations can determine a precise interpretation if the target notation has a well-defined semantics. However, this procedure is rather operational and does not provide an intuitive and abstract meaning to scenario specifications. Subtle aspects of the semantics may be, and usually are, buried in the synthesis algorithm. Some approaches translate scenario specifications into statecharts

(e.g. [72, 70, 66, 29]). In these cases, it is important to distinguish between the several different interpretations of statecharts that exist [73, 18, 74]. Furthermore, approaches that build individual statecharts for each component do not always explain how these statecharts are to be composed to provide the overall system behaviour. Other efforts based on translation have focused on producing SDL specifications (e.g. [75]), hierarchical state machines (e.g. [76]) and other state machine based formalisms (e.g. [77]).

Finally, the third category includes work in providing a formal abstract semantics for scenario-based languages. The difference to the previous category is that the semantics is defined in an abstract manner rather than by a translation algorithm. Formalisation work includes the use of process algebras [8, 10], partial orders [21], pomsets [9], buchi automata [78] and petri-nets [7]. In many cases synthesis algorithms are also provided but not as a means for defining scenario semantics. In some cases, they are developed for producing the input of model checkers [12].

Design vs. Requirement Oriented Semantics

Although the differences among approaches to scenario semantics can be considered a technical issue, the choice of mechanisms can strongly impact the role of scenario-based specification within the development process. Consequently, it also impacts on practical and research issues. There are two fundamentally different approaches to scenario semantics.

Many authors consider scenario specifications to describe high-level design of system components. In other words, that a scenario specification directly determines a state machine for each system component [68, 75, 8, 70, 66, 10, 67, 13, 29]. These approaches take on a design-oriented perspective, in which scenario descriptions are a design document in their own right.

A different interpretation of scenario specifications considers them to be describing

system functionality. In other words, that a scenario determines a set of acceptable behaviours for which many designs could be found. This view is taken by many informal approaches to scenarios (e.g. [3]) as well as approaches using semantics based on partial ordering of events [21, 61, 11]. This perspective introduces the problem of finding a design (or possibly many) for a given scenario specification and proving that the design satisfies the specifications requirements [21]. Compared to the design-oriented approach discussed previously, this approach seams to be more suitable in a requirements-oriented view phase.

2.2.4 Analysis

A crucial point in the development of concurrent systems is pre-development and pre-deployment reasoning about system requirements and design. Therefore, providing scenario-based languages with adequate syntactic and semantic constructs for specifying requirements is not enough. Scenario-based approaches must provide tools for analyzing such specifications. Unfortunately, the amount of efforts in this direction has been significantly less than the number focused on notations, features and semantics. Nevertheless, there are a number of efforts in producing techniques for analysis of scenario- based specifications, many of which focus on checking for syntactic consistency (e.g. [61, 6]). These checks depend on the features included in the sequence charts and can usually be prevented using appropriate editing tools. Some examples of what may be checked are that:

- Message labels are used consistently with respect to message initiator, recipients and parameters
- Nodes in hMSCs refer to existing bMSCs
- Timers have corresponding timeout

However, when dealing with concurrent systems, syntactic correctness is not enough and analysis of semantic implications of specifications is crucial. Many authors have focused on specific system properties and produced ad-hoc algorithms that can check their validity. For example, in sequence chart specifications that use asynchronous communication, a component can flood another by sending it an unbounded number of messages that the receiver cannot manage to process. This situation, called process divergence, can be detected using syntactic checks [11] (i.e. no translation to a semantic model is needed), and have been implemented in the MESA tool [79]. Another property detected by the same tool is that of non-local choice [11]. When using hMSCs, a given choice of sequence charts determined by the hMSC may be under-specified if the initial events of the different alternatives correspond to different components. In these cases, for all components to agree on the scenario to follow, a distributed choice among several components is needed. Race conditions where queues introduce or fail to enforce event orderings determined by scenarios have been studied by [61]. Finally, general properties based on recognising patterns of communication have also been studied [62]. Our work also addresses race and local choices but it is more focused on another property called realizability which we define formally in Chapters 5 and 6. This is complementary to other efforts in analysis and verification of scenario-based specifications. However, in contrast to other approaches to property verification, the violation of realizability is not an indication of an incorrect specification, it signals an aspect of the specification that needs to be further elaborated.

We now present a short discussion explaining where the approach we present stands with respect to the background we have presented.

Since, the aim of this thesis has been to develop automated techniques for a more reliable scenario-based software development, we have chosen to focus on sequence charts that, in addition to being widely accepted, have the advantage over free text based

scenario notations of being an intuitive and graphical representation that is amenable to computer processing. From the range of sequence chart notations that exist, we have chosen basic Message Sequence Charts (excluding advanced features such as queues, time delays, timeouts and deadlines, dynamic creation and termination of components and data). Although there is benefit in extending the expressiveness of sequence charts, this mitigates against their simplicity and intuitiveness; features that have been crucial in the wide adoption of sequence chart notations in industry. In addition, as we demonstrate in this thesis, there is much benefit to be gained even when using a very basic sequence chart notation.

The notion of scenario specifications as partial specifications is central to our choice of sequence chart semantics. If scenario specifications are partial, each scenario is simply an instance of system execution. Thus, the notion of execution is central in the notation's semantics.

From the different approaches to specifying the relation between sequence charts, we use state identification and hMSCs. While we prefer to stay with requirements-oriented abstract semantics for sequence charts, using of state identification approach enables us to enrich the scenario specification with the early feedback from the design phase even without building component's statecharts. We also use hMSCs for composing scenarios and construct scenario specifications.

We have chosen to interpret messages as asynchronous communication between components since asynchronous communication in sequence charts is more general and realistic for practical purposes. However, this comes with its own price in terms of the complexity class of the algorithm that we will be presenting in Chapters 5.

In terms of analysis there are also some differences between our approach and existing ones. Scenario specifications naturally result in partial descriptions. They depict a series of examples of what the system is expected to do. In the best of cases, these

examples cover most common system behaviours and main exceptions. This partial and incomplete nature of scenarios will be inherent both in the behaviour of individual components of a system and in the system behaviour itself. However, with regard to addressing these issues, existing works either focus on analyzing the system behaviour using some realizability notions and ignore how individual components behave, or vice versa. In contrast, we will address these separately studied problems under an integrated approach, where theoretically, for both cases the problem is sought in the behaviour of components. In regard to practical implications, we believe that analyzing both the system and the components behaviours will be more effective for elaborating scenario-based specifications.

Part I

EMERGENT BEHAVIOURS FOR THE COMPONENTS OF SCENARIO-BASED SPECIFICATIONS

Chapter 3

FROM SCENARIOS TO STATE MACHINES: INFERRENCE OF IDENTICAL STATES

Scenario-based specifications are one of the popular ways for describing concurrent systems. To a great extent, the advantage of using scenarios relates to the desire of stakeholders to describe system's functionality by small and partial stories of the system's usage. However, because each scenario gives only a partial story of system, having methodologies that can systematically derive the whole behaviour of system's components from their partial interactions in the scenarios help the designers in understanding component's behaviour. Conventionally, the process of going from a scenario specification to state machine behaviour models of components is called behaviour modeling or synthesis of state machines from scenarios.

In this chapter, we present a new approach for inferring identical states of system's components as the main activity in the synthesis of state machines from scenarios. In this approach, we assign specific *state values* to the states of system's components in different scenarios. State values are assigned using a light domain theory (see Section 3.3.1) inferred from the domain knowledge, and are used to detect identical states of components. These identical states are the place where partial behaviours from different scenarios can be merged.

The domain theory will be systematically constructed by requesting the domain expert to search through a set of pairs of messages from scenarios in order to find the pairs with a special relation called semantical causality. Semantical causality captures an invariant property of a system in terms of dependency of a message on another one, which is not

explicitly defined in scenarios.

We begin our formal definition of scenarios in this chapter. In order to be consistent with related works [21, 26], in this chapter and for the rest of the thesis, the term "processes" will be used instead of "components in scenarios".

The chapter includes 5 sections. In Section 3.1, we give a background on the synthesis problem and related works. Section 3.2 presents our definitions for scenarios and state machines. Domain theory and state values will be discussed in Section 3.3. Section 3.4 presents our definition for identical states, the domain theory director, and an example that shows how to detect identical states for an ATM system. Finally, a discussion of advantages and disadvantages of our approach compared to related works will be provided in Section 3.5.

3.1 Background

One of the problems that is usually encountered for a scenario-based specification such as Message Sequence Charts (MSC) [8], is its distributed implementation (also called realization) in terms of parallel execution of behaviour models of individual processes (or system's components) represented by state machines. In this regard, we can distinguish two different groups of works that are common in addressing this problem but are different in their assumptions and approaches.

The first group assumes that while a scenario specification is complete with respect to the behaviour of individual processes, it might not be complete with respect to the system's behaviour [21, 27, 80]. Some common problems addressed by this group are deadlocks, implied scenarios, and safe realizability. In Chapters 5 and 6 we will discuss this direction in more details.

The second group assumes that a scenario description might not be complete with

respect to the behaviour of individual processes, and therefore in the first place, they are interested in building the right behaviour models for processes using scenarios and the domain knowledge [81, 82, 30, 22, 28, 29]. A common problem addressed by this group is the emergent behaviours introduced in the behaviour models, a phenomenon that is also studied as overgeneralization.

Along with the direction of the second group, in this chapter we devise an approach for generating state machine designs (behaviour models) from scenarios. Comparing to other works, our approach has two main differences whose advantages will be discussed in Section 3.5.

First, we build a light domain theory using a given set of scenarios and the domain knowledge. The domain theory will be systematically constructed by requesting the domain expert to search through a set of pairs of messages from scenarios in order to find the pairs with a special relation called semantical causality. The semantical causality relation that will be defined in this chapter is an invariant property for a system that is not explicitly defined by scenarios and captures the dependence of a message on other messages.

Second, since detecting identical states of a process in different scenarios is necessary in the synthesis of behaviour models [30, 22, 28, 29], with the help of the constructed domain theory we use some of the messages that a process has already sent or received in order to assign a *state value* to the current state of the process. Then, a pair of states will be identical if they have the same state values.

3.2 Definitions

We assume that scenarios are represented by Message Sequence Charts defined as follows. Let P be a finite set of processes (with the total number of processes $|P| \geq 2$) and C be a

finite set of message contents (or message labels). Denote $\Sigma_i = \{i!l(c), i?l(c) | l \in P \setminus \{i\},$ $c \in C\}$ to be the alphabet of process $i \in P$, where $i!l(c)$ denotes an event that sends a message from process i with content c to process l, whereas $i?l(c)$ denotes an event that receives on process i a message with content c from process l. Also, the alphabet (of all processes $i \in P$) will be $\Sigma = \bigcup_{i \in P} \Sigma_i$.

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

- E is a finite set of events.

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

- $\beta : F! \rightarrow E?$, $F! \subseteq E!$, is a bijection mapping between send and receive events such that whenever $\beta(e_1) = e_2$ and $\alpha(e_1) = i!l(c)$, then $\alpha(e_2) = l?i(c)$.

- $\prec$ is a partial order on E such that for every process $i \in P$, the restriction of $\prec$ to E_i is a total order, and $\prec$ is equal to the transitive closure of $\{(e_1, e_2) | e_1 \prec e_2, \exists i \in P : e_1, e_2 \in E_i\} \cup \{(e, \beta(e)) | e \in F!\}$.

As an example, consider MSC1 in Fig. 3.1. A representation of this MSC in terms of Definition 1 is:

- P=$\{C1, C2\}$

- C=$\{x\}$

- $\Sigma_{C1} = \{C1!C2(x)\}$

- $\Sigma_{C2} = \{C2?C1(x)\}$

- Σ=$\{C1!C2(x), C2?C1(x)\}$

- E=$\{e_1, e_2, e_3, e_4\}$

- $\alpha(e_1) = C1!C2(x)$, $\alpha(e_2) = C1!C2(x)$, $\alpha(e_3) = C2?C1(x)$, $\alpha(e_4) = C2?C1(x)$

- E_{C1}=$\{e_1, e_2\}$, E_{C2}=$\{e_3, e_4\}$, $E!$=$\{e_1, e_2\}$, $E?$=$\{e_3, e_4\}$

- $F! = E!$, $\beta(e_1) = e_3$, $\beta(e_2) = e_4$

- $\prec$ =$\{(e_1, e_2), (e_1, e_3), (e_2, e_4), (e_3, e_4)\}$

In particular, note that $\prec$ is a partial order of E=$\{e_1, e_2, e_3, e_4\}$ since for instance, neither $(e_2, e_3) \notin \prec$ nor $(e_3, e_2) \notin \prec$.

Usually, pMSCs are restricted to a FIFO condition, which means that for all $e_1, e_2 \in E!$, if $e_1 \prec e_2$, $\alpha(c) = i!l(c)$, $\alpha(e_2) = i!l(d)$, and $e_2 \in F!$, then also $e_1 \in F!$ and $\beta(e_1) \prec \beta(e_2)$. Moreover, if for a pMSC m there exist no unmatched send events, which means $F! = E!$, then m is called a *Message Sequence Chart* (*MSC*) over P and C. Fig. 3.3 shows a set of scenarios for an ATM machine in MSC notation.

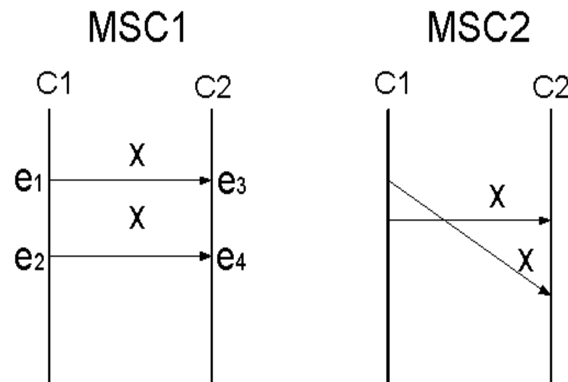


Figure 3.1: If queues are non-FIFO, there is no way for process C2 to know whether it is receiving the first or the second message x.

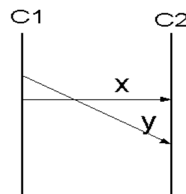


Figure 3.2: Non-FIFO condition allows for crossing of dissimilar messages.

The FIFO condition provides a unique representation for each pMSC in terms of Definition 1. This can be illustrated using Fig. 3.1. In this figure, while the message

arrows for MSC2 are crossing and as the result MSC1 and MSC2 are different, with a non-FIFO queue between C1 and C2, MSC1 and MSC2 would be the same for C1 and C2 since there is no way for process C2 to know whether it is receiving the first or the second message x. If however, we assume a FIFO queue between C1 and C2, then only the structure represented by MSC1 would be a valid pMSC and C2 always receives the first message (x) sent by C1 before the second identical message (x).

Note that, assuming FIFO queues is a stronger condition than what might be actually needed for preventing crossing of similar messages like the one in Fig. 3.1. In fact, FIFO condition does not allow for crossing dissimilar messages between processes as the one in Fig. 3.2. Because of this, some works (cf. [21]) have defined a weaker condition over pMSCs called *degeneracy* that while prevents from crossing of similar messages, it allows for the situation of Fig. 3.2. An MSC is non-degenerate if the order of receiving of two similar messages between two processes is the same as the order of their sending. More formally, non-degeneracy condition holds when for all $e_1, e_2 \in E!$, if $e_1 \prec e_2$, $\alpha(e_1) = \alpha(e_2)$, and $e_2 \in F!$, then also $e_1 \in F!$ and $\beta(e_1) \prec \beta(e_2)$. It can be seen that non-degeneracy is the restriction of FIFO condition to similar messages. Nevertheless, in this thesis we assume that the FIFO restriction holds because non-degeneracy condition has yet to be accepted as a standard for communication between processes.

Definition 2 *(Syntactical causality): For a pMSC $m = (E, \alpha, \beta, \prec)$ and $e' \in E$, define the set $S_{e'} = \{e | e' \prec e : e \in E\}$ to be the set of events in m that must happen after e' as defined by $\prec$. Then, we say e' is a syntactical cause for $S_{e'}$ and denote it by $e' \overset{sy}{\rightarrow} S_{e'}$.*

When e' is a syntactical cause for $S_{e'}$ in m and e' is removed from m, none of the events in $S_{e'}$ can happen.

We also define the projection $m|_i$ for process i in pMSC m to be the ordered sequence of messages that corresponds to the events occurring on process i in the pMSC m. For $m|_i$,

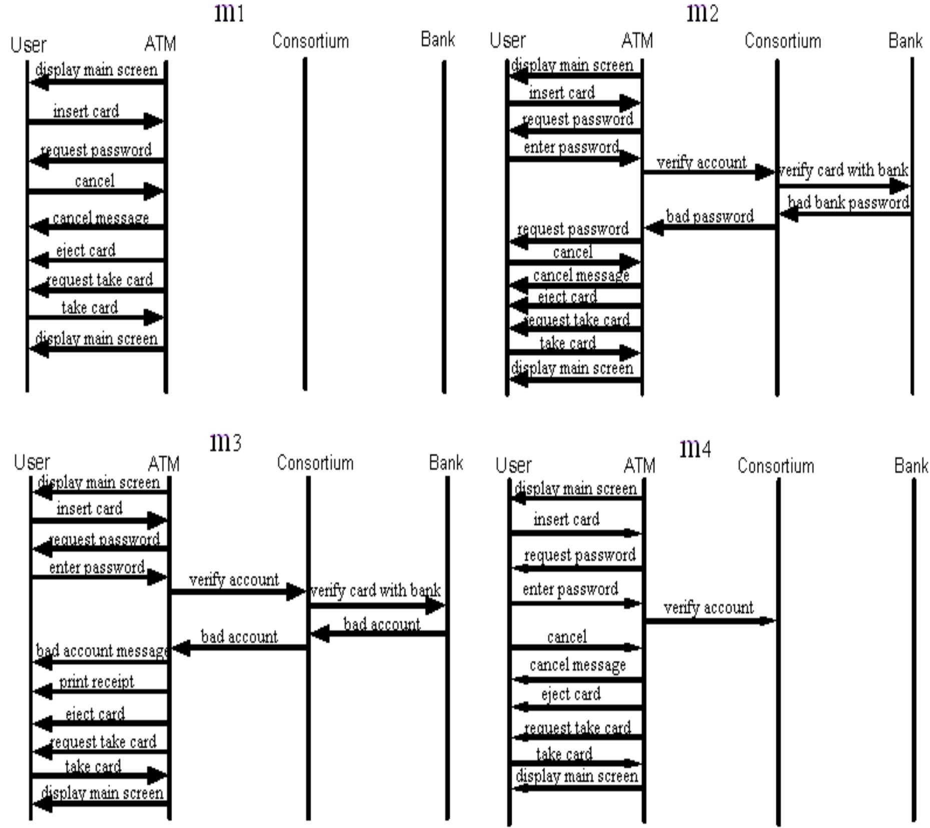


Figure 3.3: A preliminary set of scenarios for an ATM system.

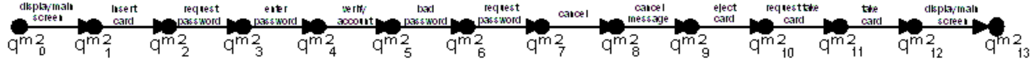


Figure 3.4: eFSM for the ATM process in m2.

$\|m|_i\|$ indicates its length, which is equal to the total number of events of m on process i, and $m|_i[j]$ refers to j^{th} element of $m|_i$, so that if e_j is the jth event on process i according to the total order of the events of i in m, then $\alpha_m(e_j) = m|_i[j-1]$, $0 < j < \|m|_i\|$.

For example, the projection of MSC1 in Fig. 3.1 on process $C1$ will be $MSC1|_{C1}$=$\{C1!C2(x)$ $C1!C2(x)\}$. Furthermore, $MSC1|_{C1}[0] = C1!C2(x)$, $MSC1|_{C1}[1] = C1!C2(x)$, and $\|MSC1|_{C1}\| = 2$.

Definition 3 *(equivalent Finite State Machine): For the projection $m|_i$, we define an equivalent FSM (eFSM) $A_i^m = (S^m, \Sigma_i, \delta^m, q_0^m, q_f^m)$ such that:*

- $S^m = \left\{q_0^m, \cdots, q_f^m\right\}$ *is a finite set of states*

- Σ_i *is the alphabet*

- q_0^m *is the initial state*

- $q_f^m = q_{\|m|_i\|}^m$ *is the final state (accepting state)*

- δ^m *is the transition relation such that* $\delta(q_j^m, m|_i[j]) = q_{j+1}^m$, $0 \leq j < f$, *and the only word accepted by* A_i^m *is* $m|_i$.

Apparently, given a projection $m|_i$, we can always construct an equivalent FSM for it. For instance, the eFSM of the projection of scenario m_2 in Fig. 3.3 on ATM process is shown in Fig. 3.4 in which $q_0^{m_2}$ is its initial state and $q_{13}^{m_2}$ is its final state.

3.3 The Domain Knowledge

3.3.1 Domain Theory

Assuming that scenarios do not provide all the necessary behaviours for processes, the domain knowledge will be needed in order to build the right behaviour models [30, 22, 29]. Here, by domain knowledge we mean the machine domain knowledge that is only pertinent to our purpose of behaviour modeling not covering all the system properties and constraints.

When dealing with the domain knowledge, a distinct feature of our approach is that it builds a domain theory that is based on an invariant property of systems called semantical causality. Semantical causality is not explicitly defined in scenarios and is different from syntactical causality of events in that syntactical causality is created because of the partial order of events in a particular scenario while semantical causality is part of system's properties. As a result, it might happen that the corresponding event of a message is a syntactical cause for the corresponding event of another message while there exists no semantical causality relation between those messages. However, the reverse is not true

i.e. whenever a message x is a semantical cause for another message y, there exists at least one scenario for which the corresponding event of x is a syntactical cause for the corresponding event of y.

Let's put semantical causality in a more formal representation and then give some examples of its usage.

Definition 4 *(Semantical causality): We say message $m|_i\ [j]$ is a semantical cause for message $m|_i[k]$ and denote it by $m|_i[j] \overset{se}{\rightarrow} m|_i[k]$, if process i has to keep the result of the operation of $m|_i[j]$ in order to perform $m|_i[k]$.*

Similar to [30], by the operation of a message we mean the ultimate purpose of the message. For example, the corresponding operation for the *insert card* message for the ATM is: *card is inserted.* Note that, semantical causality comes from the domain knowledge and can be found without referring to ordering of messages in scenarios. Also, note that it is the system's architecture and the domain knowledge that dictate whether or not one message is needed by a process in order to perform another message.

For example, in m_2, *insert card* is a semantical cause for *eject card* because ATM has to keep the card inserted before it can eject the card. As another example, in a lift system, the message *close door* is a semantical cause for the message *lift moving* because the lift has to close the door when it starts moving and keep it closed during its movements.

Now, based on Definition 4 we define the domain theory for a set of MSCs M.

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

In Section 3.4, we will explain how the domain theory can be constructed for a system.

3.3.2 State Value

One of the main problems in behaviour modeling is how to detect similar states for a process in different scenarios. The work of [29] uses a number of arbitrary system variables to distinguish the states that a process goes through as it is communicating with other processes in a scenario. However, the problem with this approach is that different domain experts can choose different system variables. Consequently, different behaviour models for a process might be obtained and at the end it is not clear which model is the right one. Note that, this problem is the result of choosing different variables not making mistakes in updating the values of variables (in fact, and implicit assumption in [29] is that the domain expert makes no mistakes in updating variables). One way to overcome this drawback is to use invariant properties of a system for evaluating and distinguishing the states of processes in scenarios. In particular, we let the current state of the process to be defined by the messages that the process needs them in order to perform the messages that come after its current state. Considering Definition 4, these are the messages that are semantical causes for the messages after the current state of the process.

More specifically, we associate a *state value* $\upsilon_i(q_k^m)$ to every state q_k^m in the eFSM A_i^m, $i \in P$, $m \in M$ as follows.

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

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

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

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

In Definition 6, first state values of the initial and the final states of an eFSM are defined. Then, the states value of the state q_k^m is defined depending on whether for the transitions that come after this state: there exists a message $m|_i[j-1]$ as a semantical cause, $0 < j < k$ (Case i)), or $m|_i[k-1]$ is the only semantical cause (Case ii)), or neither $m|_i[k-1]$ nor any other message is a semantical cause (Case iii)). In particular, the last case marks all the states q_k^m that from the processes perspective are like its initial state with the state value of 1.

Note that, since for a given application and process, semantical causality between messages is an invariant property defined by the domain knowledge, state values of the states of a process are independent of the choice of the domain expert. Also, while this definition seems to have a bit of mathematical complexity, nevertheless, it makes the process of constructing the domain theory easier for the domain expert (see Section 3.4.2).

Furthermore, unlike the approach of [29] for which only the values of system variables (not their order) are effective in distinguishing states, ordering of messages in a scenario is taken into account in state value definition in order to distinguish between states of a process. This fact can be illustrated using Fig. 3.5. Assuming that messages x and y are both changing some system variables given by an expert and also are semantical causes of message z, the values of the system variables for process $C1$ after message y in Fig. 3.5a) are the same as their values after message x in Fig. 3.5b). However, the state value of process $C1$ after message y in Fig. 3.5a) is yx, which is not the same as its state value after message x in Fig. 3.5b), which is xy (cf. Case i) of Definition 6).

As an example, let's calculate state values of states q_3^{m2} and q_7^{m2} in Fig. 3.4. From the domain knowledge pertinent to the ATM system, the maximum index j for which $m2|_{ATM}[j-1]$ is a semantical cause for a message in the transitions after q_3^{m2} is $j=2$ for which $m2|_{ATM}[2-1] =$*insert card* (for instance *insert card*$\overset{se}{\rightarrow}$*eject card*). Thus, based

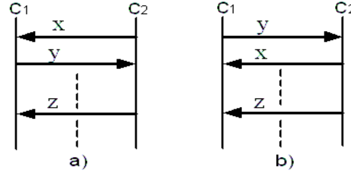


Figure 3.5: A case where state values can distinguish states of process $C1$ in a) and b), while global system variables cannot

on Case i) of Definition 6 we have: $\upsilon_{ATM}(q_3^{m2}) = m2|_{ATM}[3-1]\upsilon_{ATM}(q_2^{m2})$. To calculate $\upsilon_{ATM}(q_2^{m2})$, observe that *insert card* is the only semantical cause for messages after q_2^{m2}, and therefore, based on Case ii) of Definition 6, we have $\upsilon_{ATM}(q_2^{m2}) =$*insert card*. Thus, $\upsilon_{ATM}(q_3^{m2})$ = *(request password) (insert card)*. For q_7^{m2}, because none of the messages *bad password*, *verify account*, and *enter password* is a semantical cause for the messages in the transitions after q_7^{m2}, still the maximum index j would be $j = 2$ for which $m2|_{ATM}[2-1] =$*insert card*. Thus, with the same reasoning as for q_3^{m2}, we have: $\upsilon_{ATM}(q_7^{m2}) = m2|_{ATM}[7-1]\ \upsilon_{ATM}(q_2^{m2}) =$ *(request password) (insert card)*.

An interesting case is the state values of the user. Since the user does not need to keep (in his memory) the result of the operation of any message in order to perform other messages, there would be no semantical causality between messages for the user. Thus, according to Case iii) of Definition 6, all the states of the user except its final state are identical with the state value of 1.

3.4 Behaviour Modeling

Synthesis of behaviour models from scenarios requires a proper definition for identical states of processes and a way for detecting and merging them. In this section we give our definition for identical states and also the way that we detect them. In Chapter 4, we present our approach for merging identical states and detecting emergent behaviours.

3.4.1 Identical States

Our approach for detecting identical states of a process benefits from an intuition from converting non-deterministic finite state machines (NFA) to deterministic finite state machines (DFA) and interpret this conversion based on the past communication history of a process as follows. When the sequence of transitions that lead to two states of a process in two eFSMs (from different scenarios) are the same, the process gets into confusion in recognizing its current state and the states that the process is experiencing in different scenarios seems to be identical. Analogously, when the sequence of transitions that lead to two states (in two eFSMs) of a process are not the same, we let the process uses its state values as a watchdog in order to discriminate between its states such that whenever two states of a process have the same state values, they will be identical for the process.

Definition 7 *(Identical states): Two states* q_j^m *and* q_k^n *of process* i*, (*m *and* n *could be the same) are identical if one of the followings holds:*
i) $j = k$ *and for* $0 \leq t \prec j$*:* $m|_i[t] = n|_i[t]$
ii) $\upsilon_i(q_j^m) = \upsilon_i(q_k^n)$

Case i) of Definition 7 represents those identical states of a process that are obtained by converting the resulting NFA of the union of eFSMs to a DFA. On the other hand, Case ii) defines those identical states that the sequences of transitions before them can be different. For instance, since for q_3^{m2} and q_7^{m2} we calculated that $\upsilon_{ATM}(q_3^{m2}) =$ *(request password) (insert card)* and $\upsilon_{ATM}(q_7^{m2}) =$ *(request password) (insert card)*, q_3^{m2} and q_7^{m2} are a pair of identical states for ATM.

Note that, identical states are different from *equivalent states* defined in automata theory (two states are defined to be equivalent if every sequence of transitions that takes one of them to a final state, takes also the other one to a final state [20]). In particular,

we see identical states of a process as the source of ambiguity for the process even though minimization of FSMs that is achieved by equivalent states is also achieved through merging identical states of eFSMs. In terms of their consequences, while equivalent states do not result in emergent behaviours, identical states can result in emergent behaviours (see Chapter 4).

3.4.2 Capturing The Domain Knowledge

Even though, we can build the complete domain theory D_i for process i, we use a *domain theory director* in order to guide the domain expert to build a light domain theory that is a subset of the set D_i defined in Definition 5. The light domain theory is obtained from a set of tables (see Tables 3.1 and 3.2) such that each table represents only one member of D_i that is necessary in detecting identical states with the same incoming transitions. The reason for considering only states with the same incoming transitions is based on the following facts. First, for the purpose of behaviour modeling we are not interested in absolute values of state values. Rather, state values are used for detecting identical states. Second, since every scenario fulfills a goal for the system, usually there is a logical connection between messages in scenarios such that the current message on a process is either a semantical cause of the next immediate message or some other messages that happen later in the scenario for the process. Based on Definition 6, this means that for a given scenario it is unlikely for a process to have identical states with state values of 1 (except its initial state). In other words, it would be unlikely to have identical states that their incoming transitions are not the same. Furthermore, since identical states with state values of 1 can have any incoming transitions, in order to detect them, the domain expert needs to consider all the states in all eFSMs for a process and build the required domain theory for them. Considering the likelihood of states with the state value of 1, this requires an effort from the domain expert that most of the time does not contribute

to the detection of identical states and for that matter synthesis of behaviour models. Consequently, we only consider identical states with the same incoming transitions, which greatly reduces the time and the effort for constructing the domain theory because tables will be constructed only for the states that are immediately after repeated messages in eFSMs or pairs of eFSMs.

The Domain Theory Director

To guide the domain expert in building the required domain theory for process i, first identical states that are defined in Case i) of Definition 7 are collected in the set S_i, and equivalent states are collected in the set E_i. This can be done simply by converting the NFA of the union of the eFSMs of process i to a DFA (for S_i), and then minimizing the resulted DFA (for E_i - see [20]). Then, the domain theory director builds some tables for those pairs of states of process i that are not members of S_i or E_i (see Tables 3.1 and 3.2 - each table is denoted by $T_i(q_k^m)$, which shows the table for state q_k^m) and for which their incoming transitions are found to be the same. Definition 6 helps in building these tables in this way. Rows of table $T_i(q_k^m)$ consist of all the messages in the transitions between state q_k^m and the initial state q_0^m of A_i^m such that the first row starts with $m|_i[k-2]$. $m|_i[k-1]$ goes to the last row of the table (because for states with the same incoming transitions and state values different from 1, $m|_i[k-1]$ always appears in state values) to be the last message that would be checked for semantical causality. Moreover, columns of the tables consist of all the messages in the transitions between q_k^m and the final state q_f^m.

After tables are built, they will be filled by the domain expert. Starting from the first row of each table, the domain expert is supposed to find the first message in a row that is a semantical cause for a message in a column (see the maximum index criterion for j in Case i) of Definition 6). Consequently, starting from the first row, by finding the first

message (if there is any) in a row that is a semantical cause for a message in a column, the table filling will end for that table.

Once all the tables are filled by the domain expert, the domain theory will be all the pairs of rows and columns for which the row is specified as a semantical cause for the column. In this way, we build a light domain theory that is a subset of the set D_i defined in Definition 5 and avoid the complication and expensive cost of building D_i.

3.4.3 Domain Theory for the ATM Example

In this section, we explain how the domain theory is built for the ATM example of Fig. 3.3. In particular, we explain how Tables 3.1 and 3.2 are constructed for the repeated *request password* message in Fig. 3.4. For Table 3.1, the first two rows are the messages in the transitions before the first *request password* and the last row is the *request password* itself. Columns are the messages in the transitions after the first *request password*. Table 3.2 is similarly built for the second *request password*. These tables then can be filled using the domain knowledge pertinent to the ATM by finding the first row that its message is a semantical cause for a message in a column. Note that, only one cell is marked in Tables 3.1 and 3.2 for the row with the *insert card* message. Thus, the final domain theory for the ATM in Fig. 3.3, consists of pairs of messages (*insert card*, *enter password*) and (*insert card*, *eject card*).

Finally, note that for all other messages that are repeated for ATM in the scenarios of Fig. 3.3, the states after them are either in the set S_{ATM} (Case i) of Definition 7) or in the set E_{ATM} (equivalent states). For instance, for *cancel message* that is common between $m1$ and $m4$, q_5^{m1} is equivalent to q_7^{m4} where q_5^1 is the state after the transition of *cancel message* in the eFSM extracted from $m1$, and q_7^{m4} is the state after the transition of *cancel message* in the eFSM extracted from $m4$.

If the equivalent and the identical states of ATM are unconditionally merged, the

behaviour model shown in Fig. 3.6 will be obtained for the ATM. However, in Chapter 4 we will present a set of criteria that need to be checked before merging identical states.

Table 3.1: A part of the domain theory for ATM ($T_{ATM}(q_3^{m2})$).

	enter password	verify account	bad password	cancel	...	display main screen
insert card	$\surd$					
display main screen						
request password						

Table 3.2: Another part of the domain theory for ATM ($T_{ATM}(q_7^{m2})$).

	cancel	cancel message	eject card	request take card	take card	display main screen
bad password						
verify account						
enter password						
insert card			$\surd$			
display main screen						
request password						

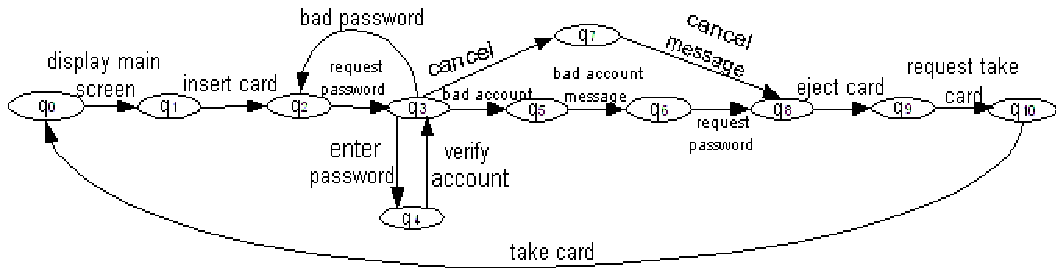


Figure 3.6: The result of merging identical states for the ATM example.

3.5 Summary

In terms of general approaches for scenario composition discussed in [28], our approach for behaviour modeling is a state identification one. In terms of using the domain knowledge, the work that is closer to our work is [29], which uses the domain knowledge in the form of OCL (Object Constraint Language) to set the values of some *global system variables* as state identifiers for processes. However, since no specific rule is given for choosing system

variables, by choosing a different set of system variables, different identical states are obtained, and hence, different behaviour models will be developed for the same process. In addition, every message in a scenario should be annotated twice with updated system variables, which needs a good deal of effort from the domain expert.

In [22], a supervised grammatical inference method ([83]) is used for synthesizing behaviour models in which the domain expert takes the role of a teacher that answers membership and conjecture queries to resolve overgeneralization in the inference process. For the membership queries, the teacher only needs to answer yes if the result of merging identical states is acceptable or no otherwise. However, to answer the conjecture queries, the teacher must be able to verify whether or not the current behaviour model is the right one. He also must be able to give proper counterexamples in the case of rejecting a conjecture. Thus, in this method instead of building a domain theory, the domain expert gives the required domain knowledge to the synthesis method.

As it is mentioned, our approach for detecting identical states of processes is a state identification one in which based on a light domain theory that we build, some of the messages that a process sends or receives are used to assign state values to the states of the process. Then, identical states are defined as the states with the same state values.

While the structure of the domain theory is simple, sometimes finding the right semantical causality between messages might be difficult in the domain theory tables. However, in such circumstances reviewing the scenario for which the table is built, can help in resolving the problem.

With respect to the domain theory tables, in general, their number is equal to the number of those repeated messages that their next immediate states in the eFSMs are not equivalent or identical by Case i) of Definition 7. Thus, the number of tables does not necessarily grows with the same rate as the size of applications. Finally, the intervention of the domain expert in the whole process is not a burden as he/she needs to (possibly)

fill only one cell in each table.

Chapter 4

FROM SCENARIOS TO STATE MACHINES: EMERGENT BEHAVIOURS

As it is stated in Chapter 3, merging identical states can result in emergent behaviours in the behaviour models. As a result, methods for the synthesis of state machine designs from scenarios must cope with two main problems, namely, generalizing partial behaviours in scenarios (discussed in Chapter 3) and a way for resolving emergent behaviours produced as the result of generalization. The challenge is that more generalization in the synthesis process may lead to more emergent behaviours and as a result, more spurious ones that are not allowed by the system architecture defined by scenarios. In this chapter, we propose a solution for this challenge in terms of a set of syntactic criteria defined over scenarios that can be automatically checked using a syntax checker in order to harness the production of spurious emergent behaviours.

4.1 Background

Synthesis of state machine designs from scenario-based specifications is a process to combine partial behaviours in the scenarios in order to obtain behaviour models. The benefit of such a synthesis process can be seen in a general software development practice, such as the one of Figure 4.1 (a simplified version of Figure 1.1). In this figure, first behaviour models are constructed from a scenario-based specification for the system. Then, an analysis phase begins in which any mismatch between the specification and the behaviour models are found in terms of emergent behaviours. After being detected, emergent behaviours will be validated against system goals and properties in order to

provide the required feedback for the system analyst to correct the specification. The process of correction and analysis continues until a satisfiable specification is reached.

Automatic synthesis of state machines from scenarios has been addressed in a number of works where a few algorithms have also been developed for this purpose ([76, 84, 81, 85, 66, 79, 22, 71, 28, 29]). A major challenge for such algorithms is that the relationship between scenarios are usually not explicitly defined. This means that the synthesis algorithm have to infer the relationship and this cannot generally be achieved without also inferring false positives or emergent behaviours [30]. These extra behaviours are not inherent to the specification and depend solely on the assumptions and the generalization technique used in the synthesis approach.

As it is mentioned before, generally, emergent behaviours are not necessary unwanted behaviours. Sometimes they may be just considered as unexpected situations due to specification incompleteness. However, some spurious emergent behaviours might also be produced as the result of generalization in the synthesis process - a phenomenon that is called overgeneralization [29]. As overgeneralization is about spurious emergent behaviours that must be resolved by the domain expert, methods for harnessing overgeneralization are an effective means in automating the synthesis process.

In this chapter, we present a method for addressing the overgeneralization problem. In particular, a set of criteria would be defined over sequence diagrams that restricts the generalization incured through behaviour modeling. Our approach has several advantages. First, because the criteria are directly defined over sequence diagrams, they can be automatically checked using a syntax checker. Second, it is virtually applicable to any approach for generating state machines as long as it starts from sequence diagrams. Third, although our approach still allows for emergent behaviours that should be reviewed and resolved by the system analyst, it prevents from overgeneralization in cases where the system architecture (including processes, messages, and the order between

events) defined by scenarios does not allow it (see Section 4.2.2).

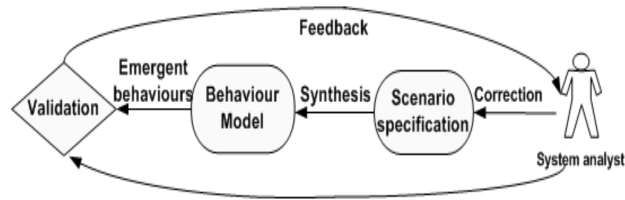


Figure 4.1: A framework for using emergent behaviours.

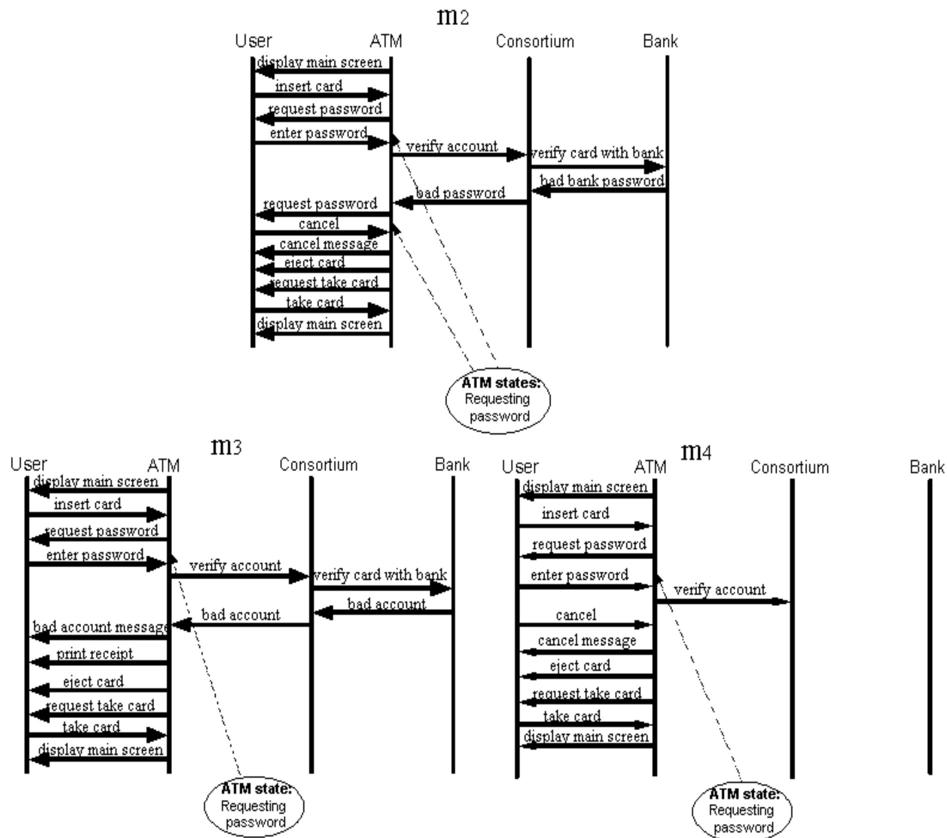


Figure 4.2: A preliminary set of scenarios for an ATM system.

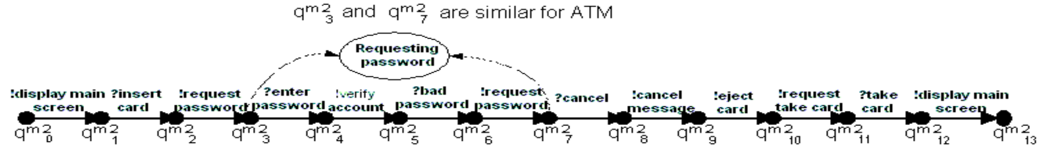


Figure 4.3: A state machine for ATM extracted from m_2.

4.2 Generalization

4.2.1 Assumptions

We assume that a preliminary set of scenarios exists and that the information regarding process's states is available (this information can be directly provided in scenarios or obtained by an approach like the one described in Chapter 3). For this purpose, we assume that only three scenarios (m_2, m_3, and m_4) of the four scenarios of Figure 3.1 are available for the ATM as depicted in Figure 4.2.

The concept of process's states can be seen in Figures 4.2 and 4.3 where the states of ATM after sending 'request password' are the same, both for the two request password messages in m_2 as well as for the request password message in scenarios m_3 and m_4. This means that from ATM's perspective, two states $q_3^{m_2}$ and $q_7^{m_2}$ in Figure 4.3 have the same state values and are identical.

4.2.2 Criteria for Merging Identical States

A common practice in behaviour modeling is to merge identical states of the process from different eFSMs [81, 82, 22, 28, 29]. In contrast, we take a different approach where we differentiate between identical states of Case i) and identical states of Case ii) as they are defined by Definition 7 in Chapter 3. This is because merging of identical states of Case i) (recall that this kind of merge is also used in converting an NFA to a DFA) does not create new behaviours for a process (the language accepted by the resulting DFA is the same as the NFA). However, merging identical states of Case ii) creates new

behaviours for a process in terms of new paths in the resulting DFA. Thus, after identical states of a process are detected, we cannot simply merge them to obtain a state machine for the process. Rather, we merge identical states if the new behaviour that could be generated as the result of this merge is allowed by the system's architecture expressed by scenarios (to be more specific, processes, messages, and the order between events). The only exception that we make is when a certain behaviour is not allowed by scenarios, but considering the domain knowledge that is used in detecting identical states, the behaviour is allowed (see Case iv) of Definition 8 and its related discussion). In this case, we let the domain knowledge override the partial order of events defined in the scenarios. In other words, between generalization and harnessing overgeneralization, the priority is given to generalization.

Figure 4.4 shows a general case where two identical states q_s and q_t of two state machines A and B for the process i are merged into a single state q. a_s, a_{s+1},..., are send or receive messages for the process from scenario m, whereas b_t, b_{t+1}, ..., are send or receive messages for the process from scenario n. Thus, $...a_s$ shows a sequence of send and receive messages that ends in a_s and $b_{t+1}...$ shows a sequence that starts with b_{t+1}.

A possible emergent behaviour in Figure 4.4 is the sequence $...a_s b_{t+1}...$ (or $...b_t a_{s+1}...$) where $...a_s$ shows a behaviour from state machine A whereas $b_{t+1}...$ is a behaviour from state machine B. In other words, the possible emergent behaviour $...a_s b_{t+1}...$ is obtained from a combination of the behaviour from A with the one from B.

Now, we shall look for criteria under which $...a_s b_{t+1}...$ is possible, where our purpose is to avoid generalization (in terms of merging states) unless checking of those criteria allows to do so.

Having said this and depending on whether b_{t+1} is a send or receive message for process i, to have $...a_s b_{t+1}...$ as the result of merging q_s and q_t, one of the following should hold:

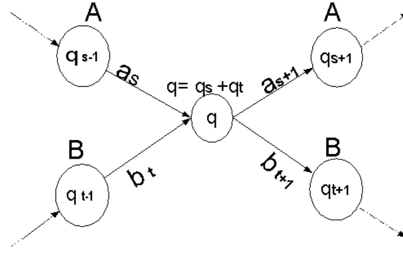


Figure 4.4: Two identical states q_s and q_t of state machines A and B are merged.

i) b_{t+1} is a send message for process i. Therefore, nothing can prevent i from sending b_{t+1} when it is in state q and generate the emergent behaviour $...a_s b_{t+1}...$

ii) b_{t+1} is a receive message for i and the following holds: in scenario m another process, say j, can send b_{t+1} to i even when a_{s+1} does not happen for it. Furthermore, in m, process i receives b_{t+1} after a_{s+1}. In this case, again the emergent behaviour $...a_s b_{t+1}...$ can happen

iii) i stops after b_t. In other words, q_t is a final state for B. In this case if a_{s+1} is a send message for i, then i has initiative to send (because of the state machine A) or stop to send (because of the state machine B) a_{s+1} when it is in state q. As a result, when the process stops sending a_{s+1}, the emergent behaviour $...a_s$ will happen. In other words, process i stops after executing the sequence of $...a_s$, while according to the scenario m and the state machine A, it must continue with message a_{s+1}

Criteria i) and iii) represent a condition where the process has initiative either to send or to stop sending a message, whereas criterion ii) represents conditions over two processes involved in sending and receiving a message. This latter case can be justified using our basic rule outlined before, that is: we look for any evidence in scenarios that allows for emergent behaviours. Because b_{t+1} is a receive message for the process, we

should look for an evidence that shows the process is able to receive b_{t+1} after a_s happens for it. Therefore, first there must be a process $j \neq i$ in m that sends b_{t+1} to i. Second, process j must be able to send b_{t+1} to i even when a_{s+1} does not happen for i because if a_{s+1} is a necessary condition to have b_{t+1} sent by j, then by removing a_{s+1}, j will not be able to send b_{t+1}. As a result, i will not be able to receive b_{t+1}, and thus, the emergent behaviour $...a_s b_{t+1}...$ will not be possible. Also, the requirement to receive b_{t+1} after a_{s+1} is to ensure that b_{t+1} is not consumed by i before a_{s+1} otherwise b_{t+1} would be simply one of the messages in the sequence $...a_s$ and it would be impossible to have the emergent behaviour $...a_s b_{t+1}....$

Note that, since the aforementioned criteria are defined over scenarios, regardless of the particular semantics of sequence diagrams, they can be automatically checked using syntactic constructs employed in the definition of sequence diagrams. These syntactic constructs include processes, events and messages, partial order between events, and similar states provided by or obtained from sequence diagrams. More specifically, for criteria i) or iii) we need respectively to check whether or not a message is a send message or a state is a final state for a given process. For criterion ii), we need to check whether a process is receiving a given message in a sequence diagram and does the partial order between events of the sequence diagram (or the state information of the sending process obtained from the domain knowledge) allows for receiving the message (see Section 4.5).

To put this in a more formal description, we can use the notation introduced in the definition of eFSMs. We assume that two identical states $q_j^m = q_s$ and $q_k^n = q_t$ of two eFSMs $A_i^m = (S^m, \Sigma_i, \delta^m, q_0^m, q_f^m)$ and $A_i^n = (S^n, \Sigma_i, \delta^n, q_0^n, q_r^n)$ (m and n could be the same) are merged into a single state q. A possible emergent behaviour is the sequence $m|_i[0]m|_i[1] \cdots m|_i[j-1]n|_i[k]n|_i[k+1] \cdots n|_i[r-1]$ where $k \neq r$, or $m|_i[0]m|_i[1] \cdots m|_i[j-1]$ where $j \neq f$ and $k = r$. These emergent behaviours are possible under the following conditions:

i) Message $n|_i[k]$ is a send message for process i. Therefore, process i has initiative to send message $n|_i[k]$ when it is in state q and we get the new sequence $m|_i[0]m|_i[1]\cdots m|_i[j-1]n|_i[k]n|_i[k+1]\cdots n|_i[r-1]$

ii) Message $n|_i[k] = i?l(c)$, $l \in P$, $c \in C$, is a receive message for process i and in MSC m process l sends a message with content c to process i ($l!i(c)$) such that process i does not receive this message before the event of $m|_i[j]$ in m and by removing the event of $m|_i[j]$ in m, still process l can send $l!i(c)$ (in terms of Definition 2 in Chapter 3, the event of $m|_i[j]$ is not a syntactical cause for the event of $l!i(c)$). In this case, again we get the new sequence $m|_i[0]m|_i[1]\cdots m|_i[j-1]n|_i[k]n|_i[k+1]\cdots n|_i[r-1]$

iii) q^n_k is the final state of A^n_i and $m|_i[j]$ is a send message for process i. Therefore, process i has initiative to send or stop to send message $m|_i[j]$ when it is in state q and we get the new sequence $m|_i[0]m|_i[1]\cdots m|_i[j-1]$

iv) Case ii) holds except that by removing the event of $m|_i[j]$ in m, process l cannot send $l!i(c)$ anymore. Then, there exists a smallest s such that the event of $m|_i[j]$ is a syntactical cause for the event of $m|_l[s]$. In this case, process l still can send message $m|_l[t] = l!i(c)$ without waiting for the event of $m|_i[j]$ to happen and create the sequence $m|_i[0]m|_i[1]\cdots m|_i[j-1]n|_i[k]n|_i[k+1]\cdots n|_i[r-1]$, if two states q^m_{s-1} and q^m_{t-1} in A^m_l will be identical (note that Case i) necessarily holds for process l in state q^m_{s-1} in A^m_l)

Note that, Case ii) for Figure 4.4 has resulted in two Cases ii) and iv). In particular, Case iv) shows a situation where sending of message $l!i(c)$ is not allowed by the partial order defined in m when the event of $m|_i[j]$ is removed. Nonetheless, a pair of identical states of process l overrides the syntactical causality defined by the partial order in m and

makes it possible for process l to send $l!i(c)$ regardless of whether message $m|_i[j]$ is being sent or not (this is the case that we allow the domain knowledge overrides the partial order of scenarios). A formal representation of the above conditions that is amenable for automatic analysis is given in Definition 8 and is called indeterministic behaviour of a process.

Definition 8 *(Indeterministic behaviour of a process): In a pMSC m, we say process i has indeterministic behaviour in state q_j^m, if there exists a pMSC n and a state q_k^n in $A_i^n = (S^n, \Sigma_i, \delta^n, q_0^n, q_r^n)$, such that q_j^m and q_k^n are identical and one of the following holds:*

i) $m|_i[j] \neq n|_i[k] = i!l(c)$ for some $l \in P$ and $c \in C$

ii) $m|_i[j] \neq n|_i[k] = i?l(c)$ for some $l \in P$ and $c \in C$, and for $\alpha(e) = m|_i[j]$, $e \in E_i$, $\exists e' \in E_l$ such that $\alpha(e') = l!i(c)$, $e' \notin S_e$ and $\beta(e') \in S_e$

iii) $r = k$ and $m|_i[j] = i!l(c)$ for some $l \in P$ and $c \in C$

iv) $m|_i[j] \neq n|_i[k] = i?l(c)$ for some $l \in P$ and $c \in C$, and for $\alpha(e) = m|_i[j]$, $e \in E_i$, $\exists e_s, e_t \in E_l$ such that: e_s is the smallest event of E_l in S_e, $\alpha(e_t) = l!i(c)$, and q_{s-1}^m and q_{t-1}^m are identical states for process l (note that, process l has also indeterministic behaviour in state q_{s-1}^m)

For FIFO queues between processes, when $m|_i[j]$ is a receive message, in Case ii) of Definition 8 it is also required that $m|_i[j]$ is sent by a process other than l. Note that out of Definition 8, it is not possible to get the new sequence $m|_i[0]m|_i[1]\cdots m|_i[j-1]n|_i[k]n|_i[k+1]\cdots n|_i[r-1]$ or $m|_i[0]m|_i[1]\cdots m|_i[j-1]$ as the result of merging q_j^m and q_k^n.

4.3 Synthesis Algorithm

Based on what we have already presented in Chapter 3 regarding identical states and considering Definition 8, Figure 4.5 shows a flowchart for synthesizing a behaviour model for process i from a set of scenarios (MSCs) M. It is assumed that for all $i \in P$ and $m \in M$, eFSMs A_i^m are available.

In the flowchart of Figure 4.5, when for two identical states q_j^m and q_k^n of process i, the process has indeterministic behaviour in q_j^m because of q_k^n, there would be an *epsilon* transition from q_j^m to q_k^n. At the end, when *epsilon* transitions are removed, the sequence of transitions after state q_k^n would be appended to the sequence of transitions before q_j^m, which can cause an emergent behaviour for the process. Accordingly, merging of q_j^m and q_k^n will be accomplished by drawing two *epsilon* transitions with opposite directions between them.

The result of applying the algorithm of Figure 4.5 to the ATM process is the same as Figure 3.6 (Chapter 3) since checking the criteria of Definition 8 results in merging all the pairs of identical states of ATM exactly the same as they were merged in Chapter 3. This fact will be illustrated in Section 4.4 for a pair of identical states ($q_3^{m_2}$ and $q_7^{m_2}$) of ATM that were also merged in Figure 3.6 (Chapter 3).

4.4 Emergent Behaviours

In this section, we show how the flowchart of Figure 4.5 can result in emergent behaviours. Consider Figure 4.3 and its two identical states ($q_3^{m_2}$ and $q_7^{m_2}$ with the same state of requesting password). Since both messages after these states are receive messages for ATM, Cases ii) and iv) of Definition 8 need to be checked. In other words, it should be checked whether or not the process (user) that sends these messages to ATM is able to send them.

The outgoing transitions from states $q_3^{m_2}$ and $q_7^{m_2}$ are respectively the enter password and the cancel messages, which are two send events for the user that occur after a pair of identical states for it (remember from Section 3.3.2 in Chapter 3 that except its final state all the states of the user have the state value of 1 and therefore, they are identical). Therefore, Case i) of Definition 8 applies to the user and the states after two request password will be merged for the user. This means that the user can either send 'enter password' or 'cancel' after receiving of 'request password'. Consequently, ATM also can either receive 'enter password' or 'cancel' after sending of 'request password'. Thus, Case iv) of Definition 8 applies for ATM for states $q_3^{m_2}$ and $q_7^{m_2}$ and these states can be merged (according to the flowchart of Figure 4.5, there would be two ϵ transitions between these two states). As the result of this merge, the state machine of Figure 4.6a) will be obtained from Figure 4.3.

Figure 4.6a) shows an emergent behaviour for ATM that is shown in Figure 4.6b) in terms of a scenario. This is a valid scenario for ATM that is ignored in the original specification given by m_2, m_3, and m_4. The system analyst (see Figure 4.1) can enrich the scenario specification for the ATM system by adding the scenario of Figure 4.6b) to the scenarios of Figure 4.2 and starts a new cycle of behaviour modeling, emergent behaviour detection, and correction.

4.5 Harnessing Overgeneralization

In Section 4.3 and through ATM example, we presented a case where an overlooked emergent behaviour was created as the result of merging two identical states for ATM after checking indeterministic behaviour for those states. In this section, we show how checking indeterministic behaviour in the flowchart of Figure 4.5 can prevent from spurious emergent behaviours that otherwise must be resolved by the system analyst.

Consider Figure 4.7, which shows a single scenario for an Alarm Clock system that is studied in [22] and assume that we want a behaviour model for the controller process. The state machine for the controller process is shown in Figure 4.8.

As can be seen in the figures, states q_1 and q_5 are identical states (because they have the same state values denoted by the label 'Displaying time'). However, this pair of identical states cannot be merged. The reason is that the outgoing transitions for both q_1 and q_5 are receive messages and hence, Cases i) and iii) of Definition 8 are not applicable. Also, the receive of 'set alarm time' is necessary for the controller to send 'register', which in turn is necessary for the timer to send 'alarm time reached' (see Figure 4.7). This means that the timer unit cannot send 'alarm time reached' to the controller unless 'set alarm time' is received by the controller. Thus, Cases ii) and iv) of Definition 8 do not hold either and as the result, q_1 and q_5 cannot be merged. Therefore, the final state machine for the controller process would be the same as the state machine of Fig. 4.8.

Note that, a synthesis method such as the one proposed in [22] (that does not have a mechanism to control overgeneralization) would output the behaviour model shown in Figure 4.9 in which q_1 and q_5 are merged and it shows the spurious emergent behaviour: display time, alarm time reached, ... (the thicker path in the figure). The path display time, alarm time reached, ... that in the first look seems a harmful behaviour ([22]) is a spurious emergent behaviour since it is not allowed by the scenario of Figure 4.7. In other words, 'alarm time reached' cannot happen for the controller until 'register' would have happened for the timer, and this message cannot happen for the timer until 'set alarm time' would have happened for the controller.

4.6 Summary

In the current practice in synthesis of behaviour models, similar states in different state machines will be merged in order to obtain a single state machine for the behaviour of each process. This practice resulted in different approaches for identifying similar states and generating state machines from scenarios [76, 84, 81, 66, 79, 22, 71, 29]. However, a common problem for these approaches is that they result in (spurious) emergent behaviours that are usually validated and resolved one way or the other by the system analyst.

The aim of this chapter was to show that looking into the information provided by scenario-based specifications is a way to assist in automatically resolving spurious emergent behaviours that otherwise puts their burden on the system analyst. Specifically, we defined a set of criteria over sequence diagrams that while still allows for generalizing partial behaviours, it prevents from overgeneralization. Furthermore, these criteria can be checked using an automated syntax checker and are independent of the particular semantics assumed for sequence diagrams such as synchronous/asynchronous, FIFO/non-FIFO buffers, etc.

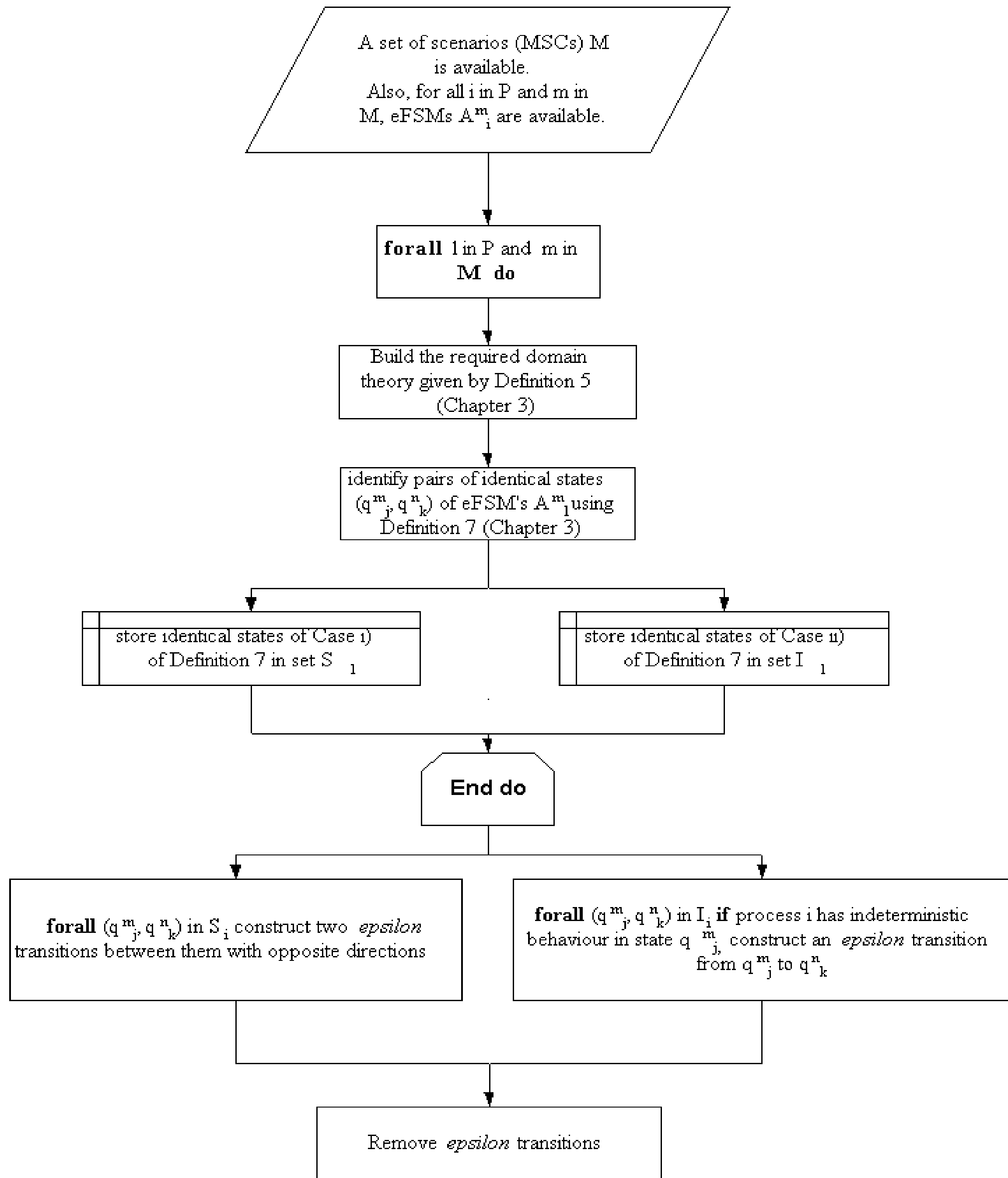


Figure 4.5: Flowchart for synthesizing a behaviour model for process i from scenarios.

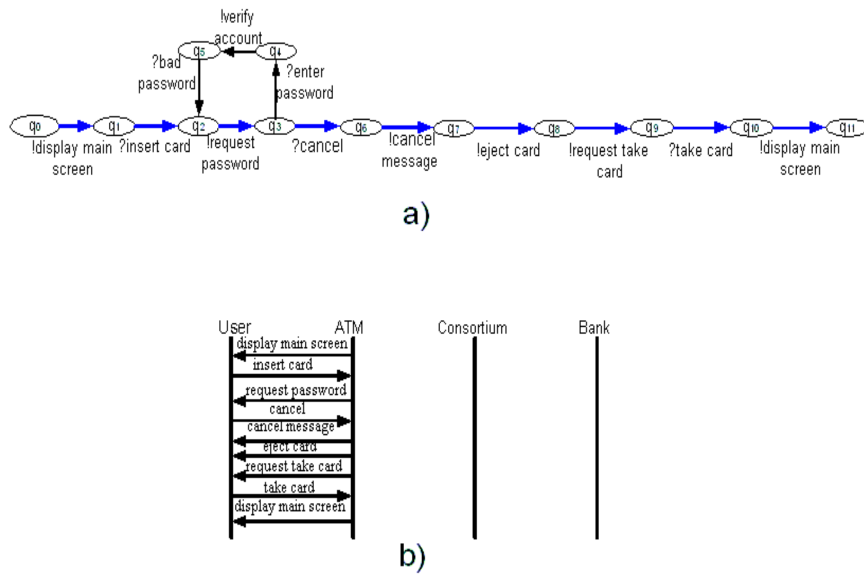


Figure 4.6: a) A state machine obtained from Figure 4.3 by merging states $q_3^{m_2}$ and $q_7^{m_2}$; b) An emergent behaviour for ATM that can be added to the set of scenarios of Figure 4.2.

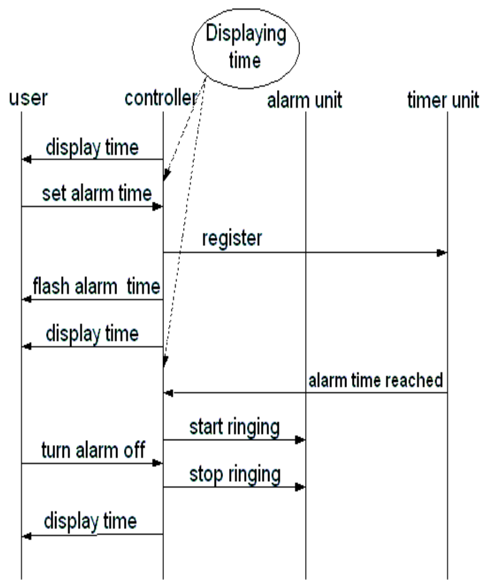


Figure 4.7: Sequence diagram for an Alarm Clock system.

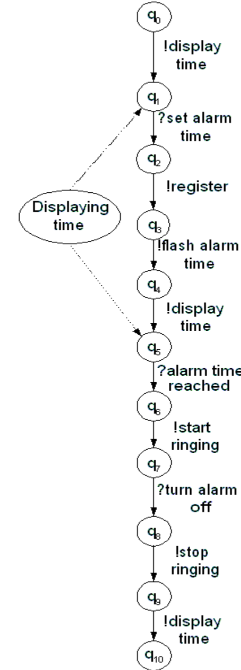


Figure 4.8: State machine for the controller process.

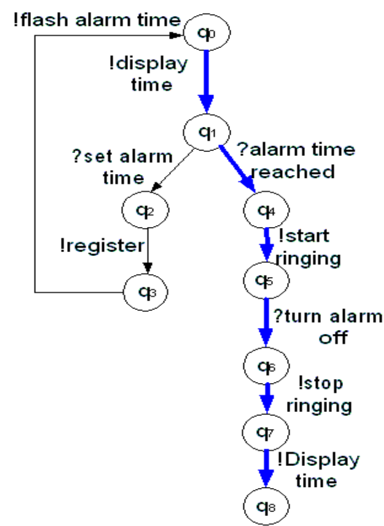


Figure 4.9: Behaviour model of the controller process when identical states are merged without checking the criteria of Definition 8.

Part II

EMERGENT SCENARIOS FOR SCENARIO-BASED SPECIFICATIONS

Chapter 5

SAFE REALIZABILITY AND IMPLIED SCENARIOS

So far, we considered the behaviour of individual processes where starting from scenarios, process's behaviour models can be synthesized. We also studied how emergent behaviours arise for processes and how they could reveal overlooked system's functionalities. Overgeneralization was addressed as a side effect of generalizing instance behaviours in scenarios and indeterministic behaviour of processes was defined in order to harness overgeneralization.

In this chapter and also Chapter 6, we consider implementation problems that are solely due to scenario specifications regardless of a particular synthesis approach used for deriving behaviour models from scenarios. In other words, even if we fix the behaviour of all processes as specified in scenarios, we want to see whether it is possible to have an implementation of the specification such that it exactly executes the behaviours specified in the specification. This problem is studied in the literature as realizability of scenario-based specifications and the behaviours that are commonly executed by all system's implementations but not specified in its scenario-based specification are called implied scenarios.

Although, there exist some methods to address and detect implied scenarios, nevertheless, there is no agreement on a well-defined and formalized cause for them. In this chapter, we will study implied scenarios and its closely related realizability notion called safe realizability. Furthermore, we show that indeterministic behaviour of processes is the cause for implied scenarios as in Chapter 4 was the cause for overgeneralization and thus, we show that these seemingly separate phenomena are created by a common cause.

Safe realizability is a measure that can say whether or not there exists a distributed

implementation of the specification such that it is deadlock free and shows exactly the behaviours specified in the specification [21, 25, 26]. A realization (or a distributed implementation) of an MSC specification is the concurrent execution of state machine models of processes. If there exists such a realization that is deadlock free and shows exactly the behaviours specified by the specification, it is said that the specification is safely realizable. If on the other hand, there is no such realization, there would be some implied scenarios that are not part of the specification but part of the behaviour of any concurrent automata covering the specification.

In this chapter, it will be shown that safe realizability of MSC specifications (in the presence of hMSCs) can be reduced to safe realizability for a set of MSCs. Based on this result, an algorithm will be presented that checks whether or not MSC specifications are safely realizable. In Chapter 6, we will extend the notion of safe realizability to address a class of emergent behaviours that are not covered by the theory of implied scenarios.

5.1 Background

Implied scenarios are a problem for distributed implementations of MSC specifications [21, 23, 27]. An implied scenario is usually defined as a behaviour that is executed by every distributed implementation of an MSC specification while it is not explicitly specified in the specification (formal definition for implied scenarios will be presented in this chapter).

The example of the process control system of Figure 5.1 and its associated scenarios in Figure 5.2 show how implied scenarios can be problematic for a system.

In this system, it is assumed that there are two chambers that are initially filled with the same gas at the same pressure and volume. Also, isothermal condition is assumed for two chambers. Each chamber has a valve connected to a pressure controller and a piston

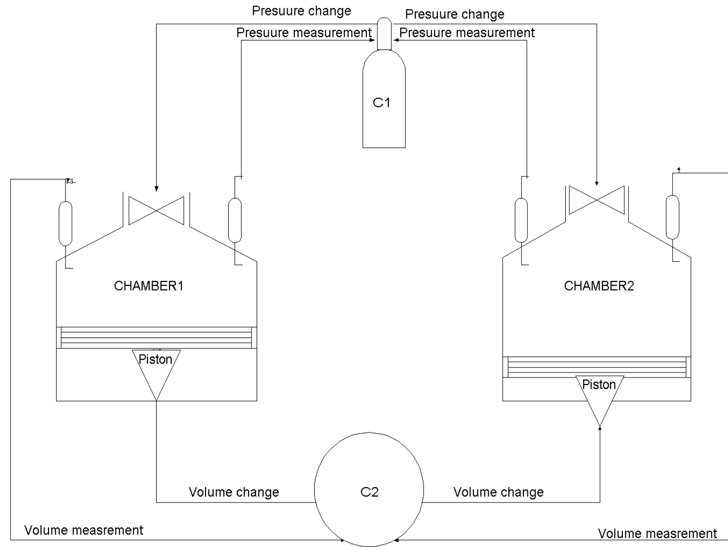


Figure 5.1: A process control system.

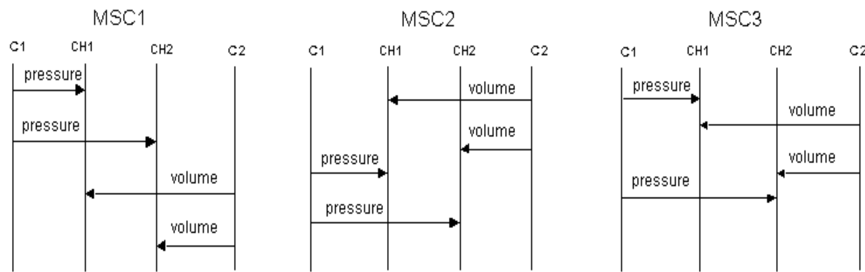


Figure 5.2: Two scenarios (MSC1 and MSC2) for the system of Figure 5.1. These scenarios can imply another scenario (MSC3) that could be unsafe for the system.

connected to a volume controller. The pressure controller can measure the pressure of each chamber and increase (or decrease) the pressure of chambers (by pumping or suction of extra gas) by a constant value (indicated in Figure 5.2 by the *pressure* message), while the volume controller can measure the volume of each chamber and increase (or decrease) the volume of chambers by a constant value (indicated in Figure 5.2 by the *volume* message). The measurement of the pressure and the volume of chambers by two controllers is irrelevant to our discussion and are excluded from Figure 5.2. For instance, Figure 5.3 shows a version of MSC1 in Figure 5.2 with messages for the measurement of the pressure

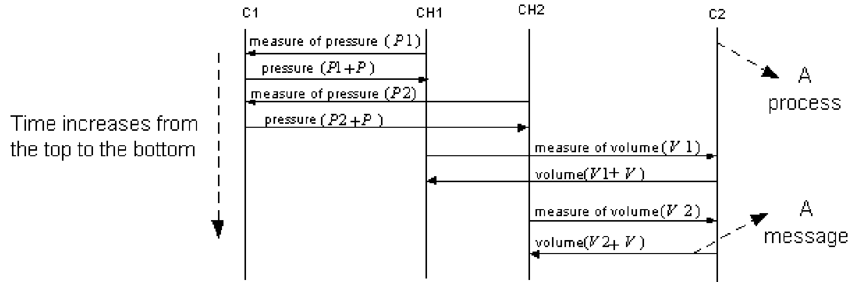


Figure 5.3: A more realistic version of MSC1, containing measurements messages.

and the volume in which the controllers send the *pressure* or the *volume* messages after having the required measurements (the *measure of pressure* and the *measure of volume* messages) from chambers.

Suppose that one of the safety requirements for this system is to have the product of the pressure and the volume of two chambers the same at the end of each scenario. In MSC1, first the pressures of two chambers are increased by the same value (P in Figure 5.3) via *controller 1 or C1* and then their volumes are increased by the same value (V in Figure 5.3) via *controller 2 or C2*. In contrast for MSC2, first the volumes of two chambers are increased by the same value and then their pressures are increased by the same value. Obviously, at the end of both MSC1 and MSC2, the product of the pressure and the volume of two chambers are the same, simply because the same action is applied in the same order to both chambers. However, based on MSC1 and MSC2 nothing can prevent a scenario like MSC3 to occur (implied by MSC1 and MSC2) because in MSC3, *C1* and *C2* have the same behaviour as in MSC1 or MSC2, *chamber 1* has the same behaviour as in MSC1, while the behaviour of *chamber 2* is the same as in MSC2. Thus, the behaviour of individual processes in this scenario is valid according to MSC1 and MSC2. However, MSC3 is an unsafe scenario for this system because the product of the pressure and the volume of two chambers are not the same when the scenario ends. This is because in MSC3, for *chamber 1 or CH1* first the pressure is increased by a constant value via *C1* and then, when *C2* increases the volume of *CH1* by a constant value, it

will certainly decrease the whole pressure of *CH1* including the constant value that is added by *C1*. But, the story for *CH2* is different since for this chamber first the volume is increased by a constant value via *C2* and then, its pressure is increased by the same constant value as for *CH1* via *C1* while this increase in the pressure will not be further changed. Another way to verify this fact is to write the gas equations for both chambers as they are going through MSC3.

5.2 Definitions

In this section, we briefly review concurrent automata, safe realizability, and implied scenarios [24].

Concurrent automata. With asynchronous message setting between processes, the behaviour of process i can be specified by an automaton A_i over the alphabet Σ_i with the following components:

1) a set Q_i of states

2) a transition relation $\delta_i \subseteq Q_i \times \Sigma_i \times Q_i$

3) an initial state $q_0 \in Q_i$

4) a set $F_i \subseteq Q_i$ of accepting states

Then, the joint behaviour of automata A_i is defined as their asynchronous product $\prod_{i \in P} A_i$ (see Section 5.4.2 for details).

The language $L(A)$ over the alphabet $\Sigma = \bigcup_{i \in P} \Sigma_i$ of the product automaton A is defined as all possible execution of A that end in an accepting state.

Scenarios as words in a formal language. In [21], scenarios are treated as words in a formal language, which is defined over the alphabet Σ. For any MSC m in a set of MSCs M, any word ω over Σ obtained by first considering a sequence of events of m that respects its partial order $\prec$, and then replacing each event by its label (as defined by the

mapping α in Definition 1 in Chapter 3) is called a linearization of m. The language $L(M)$ of M consists of all the words ω over Σ for which there exists an $m \in M$ such that ω is a linearization of m. Also, similar to the projection of MSCs on processes, for a word $\omega \in L(M)$ over the alphabet Σ its projection $\omega|_i$ on process i is defined to be the subsequence of ω that involves the send and receive events of process i.

A set of MSCs M is said to be safely realizable iff there exists a concurrent automata $A = \prod_{i \in P} A_i$ such that A is deadlock free and $L(M) = L(A)$ where a deadlock is defined as follows: A reachable state q of the product $A = \prod_{i \in P} A_i$ is said to be a deadlock state if no accepting state of A is reachable from q. Implied pMSCs on the other hand are those pMSCs that violate safe realizability of MSC specifications. Formally, a word ω' over the alphabet Σ is called an implied pMSC for a set of MSCs M iff for all product automata $A = \prod_{i \in P} A_i$ for which $L(M) \subseteq L(A)$, either ω' leads to a deadlock state in A or $\omega' \in L(A)$ and $\omega' \notin L(M)$ [21, 26, 27].

An alternative definition for implied pMSCs is based on the projections of the MSCs in M. Let $prefix(L(M))$ denotes the set of prefixes of all the words in $L(M)$. Then, we can give the following definition for implied pMSCs:

Definition 9 *Implied pMSCs. A word ω over the alphabet Σ is an implied pMSC for a set of MSCs M if one of the following holds:*

- $\forall i \in P$, $\exists m \in M$, *such that* $\omega|_i$ *is a prefix of* $m|_i$ *and* $\omega \notin prefix(L(M))$
- $\forall i \in P$, $\exists m \in M$, *such that* $\omega|_i = m|_i$ *and* $\omega \notin L(M)$

Note that, even though the prefix relation is reflexive, we might have that $\omega \in prefix(L(M))$ but $\omega \notin L(M)$. Thus, the definition is given by two conditions.

Having implied pMSCs defined, safe realizability ([21, 26]) can be defined as follows:

Definition 10 *Safe realizability for a set of MSCs. A set of MSCs M is said to be safely realizable iff there exists no implied pMSCs for M.*

5.3 Implied pMSCs and Indeterministic Behaviour of Processes

In this section, we explain how implied pMSCs are related to the process's behaviour. Scenarios are expressed here in MSC notation as the requirements specifications. Also, state machines are used as the behaviour models of processes whose concurrent execution models system's behaviour.

In Chapter 4, it was shown that emergent behaviours for processes are created only if Case ii) of Definition 7 (in Chapter 3) holds i.e. there exists a pair of identical states with the same state value. On the other hand, when studying the realizability of MSC specifications, no emergent behaviour is allowed for processes. Therefore, using Case i) of Definition 7 in Chapter 3, we can represent Cases i), ii) and iii) of Definition 8 in Chapter 4 as:

Definition 11 *(Indeterministic behaviour of a process when Case i) of Definition 7 in Chapter 3 holds for a pair of identical states): In a pMSC $m = (E, \alpha, \beta, \prec)$, we say process $i \in P$ has indeterministic behaviour because of another pMSC n, if one of the following holds:*

i) $m|_i[l] = n|_i[l]$ for $0 \leq l < k \leq \|m|_i\|$, and we have $m|_i[k] \neq n|_i[k] = i!j(c)$, for some $j \in P$ and $c \in C$

ii) $m|_i[l] = n|_i[l]$ for $0 \leq l < k \leq \|m|_i\|$, $m|_i[k] \neq n|_i[k] = i?j(c)$, for some $j \in P$ and $c \in C$, and for $\alpha(e) = m|_i[k]$, $e \in E$, $\exists e' \in E$ such that $\alpha(e') = j!i(c)$, $e' \notin S_e$ and $\beta(e') \in S_e$

iii) $m|_i[l] = n|_i[l]$ for $0 \leq l < \|n|_i\|$, and we have $m|_i[\|n|_i\|] = i!j(c)$, for some $j \in P$ and $c \in C$

As a result, in this chapter and also Chapter 6, instead of Definition 8 in Chapter 4 we use this definition for indeterministic behaviour of processes. Especially, Cases i) and ii) of Definition 11 are discussed in this chapter while Case iii) will be addressed in Chapter 6.

Now, consider Figure 5.4, which shows two scenarios in MSC notation with three processes C1, C2, and C3. Figure 5.5 shows two state machines for describing the behaviour of process C1 in MSC4 and MSC5, and Figure 5.6 is the union of two state machines of Figure 5.5.

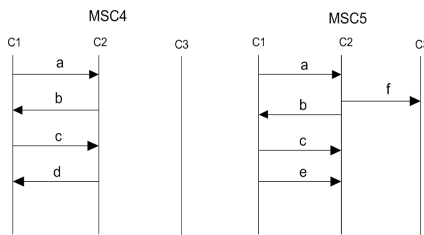


Figure 5.4: Two MSCs.

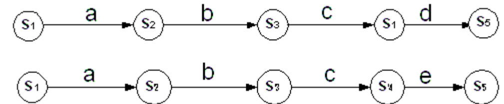


Figure 5.5: Two state machines for describing behaviour of process C1 in two scenarios of Figure 5.4.

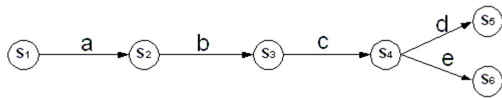


Figure 5.6: The union of two state machines of Figure 5.5.

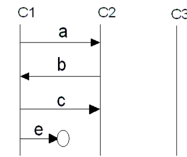


Figure 5.7: An implied pMSC obtained from MSC4.

In Figure 5.6 and from C1's perspective, after sending message c, there exists no rule to tell the next event that must happen on process C1. The immediate consequence of this lack of rule for process C1 is that starting by MSC4 and after message c, C1 can send message e instead of what is supposed to do in MSC4, which is the receive of message d. This choice of action for a process is what we defined as Case i) of Definition 11. Because of the indeterministic behaviour of C1 in MSC4, an implied pMSC will be obtained from MSC4 which is depicted in Figure 5.7 (note that this scenario in not a prefix of MSC4

or MSC5). The circle at the head of an arrow indicates that the receive part of that message may not exist.

As another case for implied scenarios consider Figure 5.8. Again, from C1's perspective and based on MSC6 and MSC7, after sending message c there exists no rule to tell the next event that must happen on this process. Also, in MSC6 there is no rule for sending message f by process C2 after receiving of message e by process C1 (there is no order between sending of message f and receiving of message e). In other words, the receive event of message e is not a syntactical cause (see Definition 2 in Chapter 3) for the send event of message f. Thus, in MSC6 message f can be sent by process C2 without waiting for the receive of message e by process C1 and the behaviour of process C1 in MSC7 gives the required certificate to process C1 to receive f instead of e. This situation is the same as what is defined as Case ii) of Definition 11.

As a result, an implied pMSC will be obtained from MSC6 that is shown in Figure 5.9.

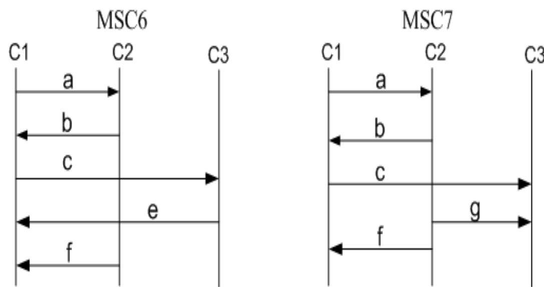


Figure 5.8: Two MSCs.

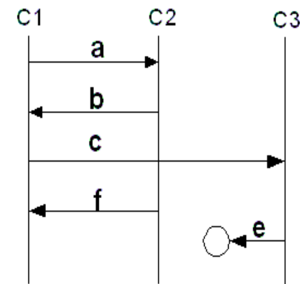


Figure 5.9: An implied pMSC obtained from MSC6.

Considering Cases i) and ii) of Definition 11 and Figures 5.4 to 5.8, we can conclude that a process i in pMSC m shows indeterministic behaviour because of another pMSC n, if for the first events e and e' on process i respectively in m and n that have different message contents, one of the following holds: i) e' is a send event (see Figures 5.4 and 5.7 and Case i) of Definition 11); ii) e' is a receive event like $i?j(c)$, $j \in P$, $c \in C$, and a

message $j!i(c)$ exists in m such that event e is not a cause for it, so that by removing e, $j!i(c)$ still can happen (see Definition 2 in Chapter 3), and this message does not have a corresponding receive event before e (see Figures 5.8 and 5.9 and Case ii) of Definition 11).

Note that, although indeterministic behaviour captures the choice of local actions for a process that is not problematic by its own, nevertheless, when the process is collaborating with other processes to fulfill a scenario, it might hinder the intended scenario to be completed.

Now, we are in a position to relate implied pMSCs to the indeterministic behaviour of processes.

Proposition 1 *If a set of MSCs M is not safely realizable, then there exist MSCs $m, n \in M$ and an implied pMSC m' such that m' is obtained from m by indeterministic behaviour of a process i in m because of n.*

Proof:

Based on Definition 10, if M is not safely realizable, M must have implied pMSCs. Suppose that the pMSC o is an implied pMSC for M. Then, remove an event e on a process $i \in P$ in o for which $S_e = \phi$ (see Definition 2 in Chapter 3) to obtain pMSC o^1, which it is only different from o in the event e. Assume that the projection of o on process i is a prefix of $n^i|_i$ for some $n^i \in M$. If e is a send event, then according to Case i) of Definition 11, process i has indeterministic behaviour in pMSC o^1 because of MSC n^i. If on the hand, e is a receive event, since o is a pMSC, the corresponding send event e' of e (as defined by β) is on a process $j \neq i$ in o. This means that e' is an unmatched send event in o^1 and thus, by Case ii) of Definition 11, process i has indeterministic behaviour in pMSC o^1 because of MSC n^i. Therefore, in either case process i has indeterministic behaviour in o^1 because of n^i.

Now, check whether $o^1 \in prefix(L(M))$. If o^1 is a prefix of $m^i \in M$, then process i has indeterministic behaviour in MSC m^i because of MSC $n^i \in M$, which results in implied pMSC $m' = o$. Thus, with $m = m^i$, $n = n^i$, and $m' = o$, the proposition is proved.

If on the other hand, $o^1 \notin prefix(L(M))$ (o^1 is an implied pMSC for M), remove an event e' on a process $j \in P$ in o^1 for which $S_{e'} = \phi$ to obtain pMSC o^2 which is only different from o^1 in the event e'. We will do the same check for o^2 as we did for o^1 i.e. if o^2 is a prefix of $m^j \in M$, then process j has indeterministic behaviour in m^j because of MSC $n^j \in M$ (assuming that the projection of process j in o^1 is a prefix of $n^j|_j$) which results in implied pMSC $m' = o^1$, and if $o^2 \notin prefix(L(M))$ (o^2 is an implied pMSC for M), remove an event e'' on a process $l \in P$ in o^2 for which $S_{e''} = \phi$ to obtain pMSC o^3 which is only different from o^2 in the event e''. If we continue with removing events in successive o^r, $0 \leq r \leq |E|$, $o^0 = o$ ($|E|$ is the number of events in o), we will get a pMSC $o^k \in prefix(L(M))$ such that $o^{k-1} \notin prefix(L(M))$ (o^{k-1} is an implied pMSC for M). This would happen since finally $o^{|E|}$ is an empty pMSC that is a prefix of $L(M)$.

Then, assuming that o^k is a prefix of an MSC $m \in M$ and $o^{k-1}|_i$ is a prefix of $n|_i$ for some $n \in M$, process i has indeterministic behaviour in m because of n, which results in an implied pMSC $m' = o^{k-1}$. This completes the proof for the proposition. $\perp$

As a result of Proposition 1, to check for implied pMSCs of M, it is sufficient to check whether indeterministic behaviour of processes in members of M result in implied pMSCs.

A straightforward (but not necessarily efficient) check for this can be done first by detecting indeterministic behaviour for a process for a pair of MSCs ($O(|E|)$), and then by checking whether this indeterminism results in an implied pMSC ($O(|M|\,|P|\,|E|)$) where $|E|$ is the number of events of the MSC that has the maximum number of events

among members of M. Considering $|P|$ processes and $|M|$ MSCs, the whole check can be done in time $O(|M|^3\,|P|^2\,|E|)$.

5.4 MSC Specifications

An MSC specification consists of a high-level Message Sequence Chart and a set of scenarios represented by Message Sequence Charts [26, 23]. A high-level Message Sequence Chart (hMSC) is a way to structure multiple scenarios (see Chapter 7 for an example of hMSCs). An hMSC $h = (V, \rightarrow, \nu_0, V_t, \mu)$ is a graph with a set of nodes V, a binary relation $\rightarrow$ over V, an initial node ν_0, an optional set of terminal nodes $V_t \subseteq V$, and a labeling function μ that maps each node to an MSC in a set of MSCs M. An hMSC h together with its set of MSCs M define an MSC specification $Spec = (h, M)$. Any sequence of nodes $\nu_0\nu_1\cdots\nu_k\cdots$ of h that starts at initial node ν_0 and $\nu_l \rightarrow \nu_{l+1}$ for $0 \leq l < k$, is an execution of h. An execution of h that is not a prefix of other executions is called a *maximal execution*. An *acceptable execution* of h either is a finite execution that terminates at a terminal node or is a maximal execution (which could be an infinite execution).

5.4.1 Some Choice Node Effects

In this section, we review some problematic choice nodes in hMSCs and show that they result in indeterministic behaviour for processes. In Section 5.4.3, a more general result will be presented which shows that the cumulative effect of all choice node properties is also captured by indeterministic behaviour of processes.

Non-local choice. A non-local choice happens when several processes independently decide to send messages to other processes [11]. An example of a non-local choice can be generated with the MSCs in Figure 5.10 by constructing a choice node from which only the MSCs BASE and NLC (Non-local Choice) can be chosen.

To formally define a non-local choice node, we assume that the following holds:

1. *Spec* = $(h,\ M)$ is normalized: for each branching node, the corresponding MSCs (as defined by μ) of its successor nodes do not have a common prefix of ordered sequence of message exchange; and

2. Each process in an MSC $m \in M$ exchanges at least one message with other processes in m

As it is explained in [11], these assumptions simplify the detection of non-local choice nodes but do not affect the generality of the results. We also use these notations. For a node ν, $succ(\nu)$ denotes all nodes that are reachable from ν through one edge and $firstprocess(\nu)$ denotes the set of processes in $\mu(\nu)$ that have the ability to send the first event in $\mu(\nu)$. Then, a non-local choice node can be defined as follows:

Definition 12 *Non-local choice node. A node ν is a non-local choice for Spec = (h, M), $h = (V, \rightarrow, \nu_0, V_t, \mu)$, if $|succ(\nu)| > 1$ and one of the following holds:*
i) $\exists \nu_k, \nu_l \in succ(\nu)$: $firstprocess(\nu_k) \neq firstprocess(\nu_l)$
ii) $\exists \nu_k \in succ(\nu)$: $|firstprocess(\nu_k)| > 1$

Now, we show that when a non-local choice exists, some processes must have indeterministic behaviour.

Proposition 2 *For any non-local choice node ν for Spec = (h, M), at least one process has indeterministic behaviour in $\mu(\nu_k)$ because of $\mu(\nu_l)$ where $\nu_k, \nu_l \in succ(\nu)$.*

Proof:

If ν is a non-local choice node for *Spec* = (h, M), $h = (V, \rightarrow, \nu_0, V_t, \mu)$, we must have $|succ(\nu)| > 1$ and one of the following holds:
i) $\exists \nu_k, \nu_l \in succ(\nu)$: $firstprocess(\nu_k) \neq firstprocess(\nu_l)$
ii) $\exists \nu_l \in succ(\nu)$: $|firstprocess(\nu_l)| > 1$

The fact that $|succ(\nu)| > 1$ shows that at least there exist two nodes $\nu_k \in succ(\nu)$ and $\nu_l \in succ(\nu)$. Then, if i) holds, it shows that there is a process i for instance in $\mu(\nu_l)$ that can send a first event in $\mu(\nu_l)$ but cannot send a first event in $\mu(\nu_k)$, which means that the first event for process i in $\mu(\nu_k)$ is different from the one in $\mu(\nu_l)$. As a result (based on Case i) of Definition 11) process i has indeterministic behaviour in $\mu(\nu_k)$ because of $\mu(\nu_l)$.

On the other hand, if ii) holds, it shows that at least two processes can send first events in $\mu(\nu_l)$. Because we assumed that h is normalized, the first event for at least one process, say i, that can send a first event in $\mu(\nu_l)$ must be different from a first event for that process in $\mu(\nu_k)$ for some $\nu_k \in succ(\nu)$. This shows (based on Case i) of Definition 11) that process i has indeterministic behaviour in $\mu(\nu_k)$ because of $\mu(\nu_l)$. Thus, the proposition is proved. $\perp$

For instance, for the non-local choice node that we built from the MSCs BASE and NLC, observe that the projection of the MSC BASE on process i has the event of sending of message a, while the projection of the MSC NLC on this process has the event of receiving of message b. Therefore, (based on Case i) of Definition 11), process i has indeterministic behaviour in the MSC NLC because of the MSC BASE.

There are some conflicts in the literature on non-local choice and implied scenarios [86]. In fact, the motivation of the work in [86] for introducing indeterministic choice and race choice is to resolve these conflicts. In the following, we review these choice node properties.

Race choice. The definition of race choice is given in [86] as follows. A node is defined to be a race choice node for a process i if the following holds: there are two different successor nodes k and l for process i such that i's first action in k is a receipt of message x and in l it is a receipt of a different message y, and such that starting with node l a

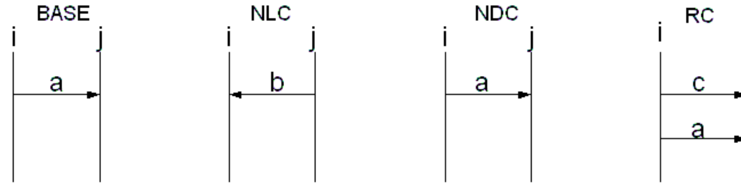


Figure 5.10: MSCs for illustrating choice nodes.

message x may be sent to i before process i performs any action.

An example of a race choice can be generated by constructing a choice node from which only the MSCs BASE and RC (Race Choice) in Figure 5.10 can be chosen. The problem with this choice node is that in the MSC RC it would be possible for process j to receive message a before message c thus producing an implied scenario.

If we compare the definition of a race choice with Case ii) of Definition 11, it shows that the former is a restricted version of the latter. In fact, Case ii) of Definition 11 gives the syntactical condition under which "message x may be sent to i before process i performs any action". But, the difference between these two is that Case ii) of Definition 11 is more general in that it is not required that message y be necessarily a receive event.

Note that, the type of queues between processes can change choice node properties. To see this for race choice, assume that FIFO queue holds between processes. In this case, if x and y are being sent by the same process (different than i), message x cannot be received by process i before message y, even though starting with node l message x may be sent to i before process i performs any action. Thus, there is no possibility that an implied scenario happens.

Indeterministic choice. An indeterministic choice node as defined in [86] is a node, which some of its successor nodes have a common first receive message. An example of indeterministic choice can be generated by constructing a choice node from which only the MSCs BASE and NDC (Non-deterministic Choice) in Figure 5.10 can be chosen.

While in [86] indeterministic choice is introduced as a separate property of choice

nodes, we do not see any reason to recognize indeterministic choice as a separate property since if the definition of race choice would not have been restricted to "receipt of different messages" (in terms of the race choice definition, the possibility of "x=y" is considered), it will cover the indeterministic choice case.

So far, we showed how choice node properties result in indeterministic behaviour for processes. In Section 5.4.3, we complete these results by showing that the cumulative effect of all choice node properties (in terms of implementation problems of MSC specifications) is also captured by indeterministic behaviour of processes.

5.4.2 Safe Realizability for MSC Specifications

To associate a language with $Spec = (h,\ M)$, we need to define concatenation of two MSCs m and m'. The concatenation of MSCs $m = (E, \alpha, \beta, \prec)$ and $m' = (E', \alpha', \beta', \prec')$ defines the MSC $m.m' = (E \cup E', \alpha \cup \alpha', \beta \cup \beta', \prec'')$, where $\prec''$ is the transitive closure of: $\prec \cup \prec' \cup \{(e, e') \in E_i \times E'_i$, for some $i \in P\}$. Then, the language $L(Spec)$ of $Spec = (h,$ $M)$ is all the MSCs (or words over the alphabet Σ) of the form $\mu(\nu_0).\ \mu(\nu_1).\ \cdots.\ \mu(\nu_k).$ $\cdots$, ν_0, ν_1, $\cdots$, $\nu_k \in V$, in which the sequence $\nu_0\nu_1\cdots\nu_k\cdots$ is an acceptable execution of h.

To define distributed implementations for MSC specifications in which hMSCs provide infinite executions for a system, we use Labeled Transition Systems (LTSs) [28], because in contrast to FSMs that only allow for finite executions, LTSs allow for both finite and infinite executions defined by hMSCs (for instance, the hMSC for the Boiler Control system given in Chapter 7 includes infinite executions for the system). An LTS T_i over the alphabet Σ_i is defined for a process i similar to an automaton (FSM) except that for an LTS the set of accepting states can be empty.

Labeled Transition System. A Labeled Transition System (LTS) over the alphabet Σ_i is defined as:

1) a set Q_i of states

2) a transition relation $\delta_i \subseteq Q_i \times \Sigma_i \times Q_i$

3) an initial state $q_0 \in Q_i$

4) a set $F_i \subseteq Q_i$ of accepting states where we can have $F_i = \phi$ (F_i can be empty)

The following algorithm shows how an LTS is obtained for process i.

Algorithm 1. Extracting process's LTSs from $Spec = (h, M)$

1. For all the MSCs $m \in M$ obtain an automaton A_i^m that accepts $m|_i$

2. For each node $\nu_l \in V$ for which $\mu(\nu_l) = m \in M$, obtain all the nodes $\nu_k \in succ(\nu_l)$, $\mu(\nu_k) = n \in M$

3. Beginning with the initial node ν_0 and for all $\nu_l \in V$, connect the accepting state of A_i^m to the initial state of each A_i^n with an ϵ transition to obtain an indeterministic LTS with ϵ transitions. If ν_l is a terminal node of h, then mark the accepting state of A_i^m as an accepting state of the LTS

4. Remove ϵ transitions and obtain a minimized LTS T_i

Note that, the LTS T_i might be indeterministic in the sense that for a pair of outgoing transitions 'x' and 'y' from a state 'q', we might have '$x = y$'.

Definition 13 *(Paths and executions of an LTS): A path $r = q_0\omega_0q_1\omega_1 \cdots \omega_{k-1}q_k$ of the LTS T_i is a sequence of states and transitions for which q_0 is the initial state of T_i and $(q_u, \omega_u, q_{u+1}) \in \delta_i$, $0 \leq u < k$. An execution of T_i is a word $\omega = \omega_0\omega_1 \cdots \omega_k$ over the alphabet Σ_i such that ω is obtained by removing all the states from a path $r = q_0\omega_0q_1\omega_1 \cdots \omega_kq_k$ of T_i. Then, we say r generates ω.*

Thus, like FSMs, any word ω over the alphabet Σ_i that brings an LTS T_i from its initial state to any state in T_i is an execution of T_i. Note that, for a given word ω there might be several corresponding paths in T_i that generate ω.

Definition 14 *(Derived paths of an LTS): A path* $s' = q_0\omega'_0q'_1\omega'_1\cdots\omega'_{k'-1}q'_{k'}\cdots$ *of the LTS* T_i *is derived from another path* $s = q_0\omega_0q_1\omega_1\cdots\omega_{k-1}q_k\cdots$ *of the LTS* T_i*, if* s' *can be obtained from* s *by repeating some sequences of states and transitions* $q_u\omega_u\cdots\omega_{v-1}q_v$ *in* s*,* $0 \leq u, v \leq k$*. The set of all paths of* T_i *that can be derived from a path* s *in* T_i *is denoted by* P_s*.*

Definition 15 *(Derived words of an LTS): We say a word* ω *is derived from a path* s *if there exists a path* $s' \in P_s$ *such that* s' *generates* ω*. The set of all words that can be derived from* s *is denoted by* W_s*.*

A path of an LTS might include loops. Formally, for a path $r = q_0\omega_0q_1\omega_1\cdots\omega_kq_k$ of the LTS T_i, a sequence $q_u\omega_u\cdots\omega_{v-1}q_v$, $0 \leq u, v \leq k$, is a loop in r if $u = v$. Loops might cause a single execution to be generated by several paths for indeterministic LTSs. But, the relation between paths and executions is one-to-one for deterministic LTSs.

Having a many-to-one relation between paths and executions for an indeterministic LTS might create a kind of redundancy for an LTS where paths that are able to generate different executions also generate common executions. Thus, we define non-redundant LTSs to capture this redundancy.

Definition 16 *(Non-redundant LTS): An LTS* T_i *is non-redundant if for any two paths* r *and* s *of* T_i *for which* $W_r \neq W_s$*, we have that* $W_r \cap W_s = \phi$ *where* ϕ *is the empty set.*

Note that, an indeterministic LTS might be redundant or non-redundant. However, any deterministic LTS is non-redundant. This fact is shown by the following lemma.

Lemma 1 *Any deterministic LTS is non-redundant.*

Proof:

Consider two paths $r = q_0\omega_0 q_1\omega_1 \cdots \omega_k q_k$ and $s = q_0\omega'_0 q'_1\omega'_1 \cdots \omega'_{k'} q'_{k'}$ of a deterministic LTS T_i. Because T_i is deterministic and r and s are different paths, $\exists u$ such that for all $0 \leq x \leq u$, we have $q_x = q'_x$, and $\omega_u \neq \omega'_u$. As a result, $q_u\omega_u$ that is a sequence in any path in P_r and $q_u\omega'_u$ that is a sequence in any path in P_s would be different. Consequently, we must have $W_r \neq W_s$ and $W_r \cap W_r = \phi$, and the lemma is proved. $\perp$

The language of an LTS T_i is defined by maximal executions of T_i. A maximal execution of T_i is an execution of T_i that is not a prefix of other executions. The language $L(T_i)$ consists of all the words ω over the alphabet Σ_i such that ω either is a finite execution of T_i that ends in an accepting state or is a maximal execution of T_i (which can be infinite).

Next, we define the asynchronous product of the LTS's T_i. Note that, the asynchronous product automata discussed in Section 5.2 is also defined similarly based on local automata A_i.

We start with the buffers between processes and for each ordered pair (i, j) of processes, two message buffers $B^s_{i,j}$ and $B^r_{i,j}$ are defined. $B^s_{i,j}$ stores the messages that have been sent by process i but are still in transit and not yet accessible by process j and $B^r_{i,j}$ stores messages that have already reached process j but are not accessed and removed from the buffer by process j. All the buffers are words over the set of message contents C. We also define the following operations on the buffers. $isTrue(B^r_{i,j}, c)$ returns **true** if $B^r_{i,j}$ contains the message c in the front and returns **false** otherwise; $append(B^{s,r}_{i,j}, c)$ appends message c to the buffer $B^{s,r}_{i,j}$; and $remove(B^{s,r}_{i,j}, c)$ deletes message c in front of the buffer $B^{s,r}_{i,j}$. Then, the product LTS $T = \prod_{i \in P} T_i = (Q, \Sigma, \delta, q_0, Q_f)$ with asynchronous message passing is defined as follows:

Q: A state $q \in Q$ consists of the local states q_i of local LTS's T_i, along with the contents

of the buffers $B^s_{i,j}$ and $B^r_{i,j}$.

q_0: The initial state q_0 of T is given by having all the local LTS's in their start states q^0_i, and by having every buffer empty.

δ: The transition relation $\delta \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ (the τ transitions model the transfer of messages from the sender to the receiver) is the smallest relation satisfying following rules:

- $\dfrac{q=\left\{q_1,q_2,\cdots,q_i,\cdots,B^s_{i,j},B^r_{j,i},\cdots,B^r_{l,k}\right\},\ \ (q_i,i?j(c),q'_i)\in\delta_i,\ \ isTrue(B^r_{j,i},c)}{q'=\left\{q_1,q_2,\cdots,q'_i,\cdots,B^s_{i,j},remove(B^r_{j,i},c),\cdots,B^r_{l,k}\right\},\ \ (q,i?j(c),q')\in\delta}$

- $\dfrac{q=\left\{q_1,q_2,\cdots,q_i,\cdots,B^s_{i,j},B^r_{i,j},\cdots,B^r_{l,k}\right\},\ \ (q_i,i!j(c),q'_i)\in\delta_i}{q'=\left\{q_1,q_2,\cdots,q'_i,\cdots,append(B^s_{i,j},c),B^r_{j,i},\cdots,B^r_{l,k}\right\},\ \ (q,i!j(c),q')\in\delta}$

- $\dfrac{q=\left\{q_1,q_2,\cdots,q_i,\cdots,B^s_{i,j},B^r_{i,j},\cdots,B^r_{l,k}\right\}}{q'=\left\{q_1,q_2,\cdots,q_i,\cdots,remove(B^s_{i,j},c),append(B^r_{i,j},c),\cdots,B^r_{l,k}\right\},\ \ (q,\tau,q')\in\delta}$

Q_f: $q \in Q_f$ if for all processes i, the local states q_i of process i in q is accepting, and all the buffers in q are empty.

The asynchronous product $T = \prod_{i\in P} T_i$ defines the language $L(T)$ that consists of all the words over the alphabet Σ that either are the result of finite executions of T that end in an accepting state or are the maximal executions of T where τ transitions are dealt with as usual ϵ transitions in automata theoretic sense.

Definition 17 *A product LTS* $T' = \prod_{i\in P} T'_i$ *is a distributed implementation of* $Spec = (h,\ M)$ *if* $L(Spec) \subseteq L(T')$.

Moreover, similar to deadlock states in the concurrent automata A, we have deadlock states in the concurrent LTS $T' = \prod_{i\in P} T'_i$. However, instead of unreachability of accepting states from a deadlock state q, we require that starting from q only finite executions are possible that none of them ends in an accepting state.

In the remaining of this section, first the following two lemmas are given, which in Section 5.4.3 are used for characterizing safe realizability for MSC specifications. Then, safe realizability will be defined for MSC specifications.

Lemma 2 *For Spec* $= (h,\ M)$*,* $h = (V, \rightarrow, \nu_0, V_t, \mu)$*, and any execution* ω *of* T_i*, there exists an execution* $\nu_0\nu_1 \cdots \nu_k$ *of* h *such that* $\omega = \mu(\nu_0).\ \mu(\nu_1).\ \cdots.\ \mu(\nu_k)|_i$.

Proof:

The proof comes directly from Algorithm 1 in which the concatenation of the automata for process i is achieved by following the corresponding MSCs in different executions of h (by starting from ν_0 and finding all $\nu_k \in succ(\nu_l)$ for each $\nu_l \in V$). $\perp$

Lemma 3 *For Spec* $= (h,\ M)$*, let* T_i *be a local LTS obtained by Algorithm 1. Then,* $T = \prod_{i \in P} T_i$ *will be the minimal distributed implementation of Spec* $= (h,\ M)$ *i.e. for any other distributed implementation* $T' = \prod_{i \in P} T'_i$ *for Spec, if* ω *is an execution of* T*, then there exists an execution* ω' *for* T' *such that* ω *is a prefix of* ω'*.*

Proof:

By Lemma 2, since ω is the result of concurrent execution of T_i's, $\forall i \in P$, we must have $\omega|_i = \mu(\nu_0).\ \mu(\nu_1).\ \cdots.\ \mu(\nu_k)|_i$ for some execution $\nu_0\nu_1 \cdots \nu_k$ of h. Now, if ω is not a prefix of executions of another distributed implementation $T' = \prod_{i \in P} T'_i$ of *Spec* for which $L(Spec) \subseteq L(T')$, it means that ω cannot be executed by $T' = \prod_{i \in P} T'_i$. This means that at least one local LTS such as T'_i cannot execute $\omega|_i$. As a result, T' would not be able to execute the MSC $\mu(\nu_0).\ \mu(\nu_1).\ \cdots.\ \mu(\nu_k)$ with the projection $\omega|_i$ on process i. But this is impossible since $L(Spec) \subseteq L(T')$, and therefore T' must be able to execute $\mu(\nu_0).\ \mu(\nu_1).\ \cdots.\ \mu(\nu_k)$. Thus, there must be an execution ω' for T' such that ω is a prefix of ω'. $\perp$

Now, safe realizability for MSC specifications can be defined as follows [23]:

Safe realizability for MSC specifications. An MSC specification *Spec* $= (h,\ M)$ is said to be safely realizable iff there exists a product LTS $T' = \prod_{i \in P} T'_i$ such that T' is

deadlock free and $L(Spec) = L(T')$.

5.4.3 Basic Executions

In this section, we characterize safe realizability in terms of executions of hMSCs. The bottom line of our approach is to characterize those executions of hMSCs that their length is as short as possible and guarantee to capture deadlocks and implied scenarios (if there is any) of *Spec*.

Definition 18 *(Basic paths of an LTS): Let $r = q_0\omega_0 q_1\omega_1 \cdots \omega_k q_k$ be a path of T_i (q_0 is the initial state of T_i). Then, r is called a basic path of T_i either if r does not go consecutively through any loop and q_k is an accepting state of T_i, or r satisfies the following conditions:*

i) r does not go consecutively through any loop

i) For all reachable states q_{k+1} from q_k, $q_{k+1} \in \{q_0, , q_1, \cdots, q_k\}$

iii) $q_{k-1}\omega_k q_k$ is not a repeated sequence in r

iv) r is not a prefix of any other path of T_i that satisfies i), ii), and iii)

Definition 19 *(Basic executions of an LTS): A word $\omega = \omega_0\omega_1 \cdots \omega_u$ over the alphabet Σ_i is called a basic execution of T_i if ω is generated by a basic path of T_i.*

In the definition of basic paths, first we ensure that an execution is extended to include as many as new reachable states. After it is found that the execution cannot be further extended, we ensure that no repeated sequence of states and transitions that

have been traversed before (which their inclusion in the path only increases its length without exploring new states/transitions), is included in the path.

Now, we are in a position to characterize those executions of the hMSC h that can determine whether the MSC specification $Spec = (h,\ M)$ is safely realizable. These executions are called *basic executions* of h.

Definition 20 *(Basic executions of hMSCs): Let* $t = \nu_0\nu_1 \cdots\ \nu_k$, $\nu_l \rightarrow \nu_{l+1}$, $0 \leq l < k$ *be an execution of* $h = (V, \rightarrow, \nu_0, V_t, \mu)$*. Then,* t *is called a basic execution of* h *either if* t *does not go consecutively through any loop and* ν_k *is a terminal node of* h*, or* t *satisfies the following conditions:*

i) t *does not go consecutively through any loop*

ii) for all reachable nodes ν_{k+1} *from* ν_k, $\nu_{k+1} \in \{\nu_0, \nu_1, \cdots,\ \nu_k\}$

iii) $\nu_{k-1}\nu_k$ *is not a repeated sequence in* t

iv) t *is not a prefix of any other execution of* h *that satisfies i), ii), and iii)*

The following lemma relates basic executions of the hMSC h to basic executions of the local LTSs T_i.

Lemma 4 *For* $Spec = (h,\ M)$*, let* T_i *be a local LTS obtained by Algorithm 1. Then, for any basic execution* ω *of* T_i*, there exists a basic execution* $s = \nu_0\nu_1 \cdots \nu_k$ *of* h*,* $m = \mu(\nu_0)\mu(\nu_1)\ \cdots\ \mu(\nu_k)$*, such that* ω *is a prefix of* $m|_i$*.*

Proof:

Consider a basic execution ω of T_i. If ω is generated by a basic path of T_i that terminates at an accepting state of T_i, then since loops in the LTSs T_i are the projections

of the loops in h, there exists an execution $s = \nu_0\nu_1 \cdots \nu_k$ of h that terminates at a terminal node and for $m = \mu(\nu_0)\mu(\nu_1) \cdots \mu(\nu_k)$, $m|_i = \omega$. Moreover, we can choose s such that it does not go consecutively through loops. This is because any loop c in s either has a projection loop c_i on T_i or not. If c has a projection c_i, then we cannot go through it consecutively, otherwise $m|_i = \omega$ will not be a basic execution of T_i. On the other hand, if c does not have any projection on T_i, then the number of traversing c has no effect on $m|_i$. Thus, $s = \nu_0\nu_1 \cdots \nu_k$ is a path in h that does not go consecutively through any loop and terminates at a terminal node of h, which based on Definition 20 would be a basic execution of h.

On the other hand, if ω is generated by a basic path of T_i that does not end at an accepting state of T_i, then still there exists an execution $s = \nu_0\nu_1 \cdots \nu_{k-1}\nu_k$ of h such that for $m = \mu(\nu_0)\mu(\nu_1) \cdots \mu(\nu_k)$, ω is a prefix of $m|_i$. Also, we still can choose s such that it does not go consecutively through any loop.

Let $r = q_0\omega_0 q_j\omega_1 \cdots \omega_u q_u$ be the basic path in T_i that generates $\omega = \omega_0\omega_1 \cdots \omega_u$. Then, since $q_{u-1}\omega_u q_u$ is not a repeated sequence in r (see Definition 18), $\nu_{k-1}\nu_k$ cannot be a repeated sequence in s.

Now, if there exists a reachable node ν_{k+1} from ν_k such that $\nu_{k+1} \notin \{\nu_0, \nu_1, \cdots, \nu_k\}$, or if for all reachable nodes ν_{k+1} from ν_k, $\nu_{k+1} \in \{\nu_0, \nu_1, \cdots, \nu_k\}$ but s is a prefix of another basic execution of h (see Definition 20), we can continue s and obtain a basic execution $s' = \nu_0\nu_1 \cdots \nu_{k'}$ of h. Then, for $n = \mu(\nu_0)\mu(\nu_1) \cdots \mu(\nu_{k'})$, ω would be a prefix of $n|_i$.

Otherwise, s must be a basic execution of h and for $\omega = m|_i$, ω would be a prefix of $m|_i$. This completes the proof of the lemma. $\perp$

Some executions of an hMSC can be derived from other execution by repeating some sequences of nodes. Formally, we can define the executions that can be derived from a

single execution as follows.

Definition 21 *(Derived executions of hMSCs): For the hMSC* $h = (V, \rightarrow, \nu_0, V_t, \mu)$*, an execution* $t' = \nu_0\nu'_1 \cdots\ \nu'_{k'}$ *of* h*,* $\nu_l{\rightarrow}\nu_{l+1}$*,* $0 \leq l < k'$*, is said to be derived from another execution* $t = \nu_0\nu_1 \cdots\ \nu_k$ *of* h*,* $\nu_l{\rightarrow}\nu_{l+1}$*,* $0 \leq l < k$*, if* t' *can be obtained from* t *by repeating some sequences of nodes* $\nu_u\nu_{u+1}\cdots\nu_v$ *in* t*,* $0 \leq u, v \leq k$*.*

The following theorem shows that any execution of h can be derived from a prefix of a basic execution.

Lemma 5 *For* $Spec = (h,\ M)$*,* $h = (V, \rightarrow, \nu_0, V_t, \mu)$*, and for any execution* t *of* h *there exists a basic execution* t' *of* h *such that* t *can be derived from a prefix of* t'*.*

Proof:

Consider an execution $t = \nu_0\nu_1\cdots\nu_k$ of h. There are 4 possibilities for t:

1) t does not go consecutively through any loop and either there exists a reachable node ν_{k+1} from ν_k such that $\nu_{k+1} \notin \{\nu_0, \nu_1, \cdots, \nu_k\}$, or for all reachable nodes ν_{k+1} from ν_k, $\nu_{k+1} \in \{\nu_0, \nu_1, \cdots, \nu_k\}$, but t is not a basic execution of h because t is a prefix of another basic execution. Then, we can add a node ν_{k+1} to the end of t and extend t to obtain a basic execution t'. In this case, t would be a prefix of the basic execution t' and thus, it can be derived from a prefix of t'.

2) t is a basic execution of h. Then, for $t = t'$, t can be derived from t'.

3) t does not go consecutively through any loop and for all reachable nodes ν_{k+1} from ν_k, $\nu_{k+1} \in \{\nu_0, \nu_1, \cdots, \nu_k\}$, and t is not a prefix of a basic execution of h. However, t still is not a basic execution of h because $\nu_{k-1}\nu_k$ is a repeated sequence in t. Then starting from the end of t, we can delete all the nodes ν_z for which $\nu_{z-1}\nu_z$, $0 < z \leq k$, is a repeated sequence in t to obtain an execution t'. Since by extending t' no other basic execution is

obtained for h, t' would be a basic execution for h. In this case, t can be derived from basic execution t' by repeating some sequences of nodes in t'.

4) t goes through a loop consecutively. Then, by deleting extra loops in t we can obtain an execution t'' of h that includes all the nodes (and their sequencing) in t except that for every loop that t goes consecutively through it, t'' goes only once. Now, t'' has one of the conditions described in Cases 1), 2), and 3) for which we showed that t'' can be derived from a prefix of a basic execution t' of h. As a result, t also can be derived from a prefix of t'. This completes the proof for the lemma. $\perp$

For $h = (V, \rightarrow, \nu_0, V_t, \mu)$, $Spec = (h,\ M)$, let B be a set whose members are all the MSCs $\mu(\nu_0)\mu(\nu_1)\cdots\mu(\nu_k)$ for which $\nu_0\nu_1\cdots\nu_k$ is a basic execution of h. Our purpose would be to build a proper relation between safe realizability of $Spec = (h,\ M)$ and safe realizability of B. In so doing, first we identify those implied pMSCs of B that cannot be implied pMSCs for $Spec = (h,\ M)$.

Lemma 6 *Let X be a set for which $\forall m \in X$, m is an implied MSC for B that is also a prefix of a member of B, and $\exists i \in P$ such that $m|_i$ is not an acceptable execution of T_i. Then, $\forall m \in X$, m cannot be an implied pMSC for $Spec = (h,\ M)$.*

Proof:

An implied pMSC for $Spec = (h,\ M)$ either results in a deadlock in the minimal distributed implementation $T = \prod_{i \in P} T_i$ or is an acceptable execution of T that is not in $L(Spec)$. We show that none of these cases are possible.

Because m is a prefix of a member n of B, we have $m \in prefix(L(B))$ where $prefix(L(B))$ is the set of all the prefixes of the members of B. Then, since $L(B) \subseteq prefix(L(Spec))$, we must have $m \in prefix(L(Spec))$. As a result, m will not result in deadlocks in T.

On the other hand, since $\exists i \in P$ such that $m|_i$ is not an acceptable execution of T_i, m cannot be an acceptable execution of $T = \prod_{i \in P} T_i$ and thus, m cannot be an implied scenario for $Spec = (h,\, M)$. This completes the proof. $\perp$

Once X has been characterized as a set whose members are those implied pMSCs of B that cannot be implied pMSCs for $Spec = (h,\, M)$, the relation between implied pMSCs and safe realizability of $Spec = (h,\, M)$ with implied pMSCs and safe realizability of the union of B and X can be established. However, the results would be different depending on whether or not the hMSC h is bounded.

In a bounded hMSC h, the communication between processes in every loop in h is performed in such a way that prevents from flooding a process by the messages sent by another process [26]. One consequence of bounded hMSCs that would be relevant to the results of this section is that after a loop of h is fully traversed, in order to start a new round through the loop, all processes that are active in the loop (a process is active in a loop if it sends or receives at lease one message in the loop) must send/receive all their messages and thus, all the buffers between processes must be empty. This means that the pattern of communication of the active processes in the loop is independent of how many times the loop will be traversed.

One consequence of the boundedness property is that since it has been proved that in asynchronous setting safe realizability for unbounded hMSCs is undecidable [24], any algorithm that decides on safe realizability of MSC specifications must assume bounded hMSCs.

Thus, in this section first we prove a general relation between implied pMSCs of the union of B and X with implied pMSCs of $Spec = (h,\, M)$ when no assumption is made regarding whether h is bounded or not. Apparently, this result also holds for bounded hMSCs. Then, we use this result for bounded hMSCs and give an algorithm that can

decide on safe realizability of $Spec = (h,\, M)$.

Theorem 1 *For $Spec = (h,\, M)$, $h = (V, \rightarrow, \nu_0, V_t, \mu)$, let T_i be a local LTS obtained by Algorithm 1. Then, if $\forall i \in P$, T_i is non-redundant, any implied pMSC of $B \cup X$ will be an implied pMSC for Spec.*

Proof:

Let's $B' = B \cup X$ and assume an implied pMSC m for B' that is not a prefix of members of B' i.e. $\forall i \in P$, $\exists n \in B'$ such that $m|_i$ is a prefix of $n|_i$ and $m \notin prefix(L(B'))$ where $prefix(L(B'))$ is the set of all the prefixes of the members of B'. We show that $m \notin prefix(L(Spec))$.

To see this, let's assume that $m \in prefix(L(Spec))$. Then, m must be a prefix of an MSC $o \in L(Spec)$, $o \notin prefix(L(B'))$. Let $t = \nu_0\nu_1 \cdots \nu_k$ be an execution of h for which $o = \mu(\nu_0)\mu(\nu_1)\cdots\mu(\nu_k)$. Then, since $o \notin prefix(L(B'))$, by Lemma 5 (see the proof), t can be derived from a prefix of a basic execution $t' = \nu_0\nu_1 \cdots \nu_{k'}$ of h with one of the following conditions:

1) t goes consecutively through a loop in t'

2) t does not go consecutively through any loop in t', however t' is a prefix of t i.e. $\forall k < u \leq k'$, $\nu_u \in \{\nu_0, \nu_1, \cdots, \nu_{k-1}, \nu_k\}$

If m is a prefix of o but not a prefix of members of B', then there must exist an $i \in P$ such that $m|_i$ is a prefix of $o|_i$ but not a prefix of $n'|_i$ where $n' \in B'$ is the MSC created by t', that is, $n' = \mu(\nu_0)\mu(\nu_1)\cdots\mu(\nu_{k'}) \in L(B')$. Also, $m|_i$ must be generated by two different paths of T_i: a path r that does not go consecutively through any loop and generates $m|_i$ as a prefix of $n|_i \neq n'|_i$ for some $n \in B'$, and another path s that generates $m|_i$ as a prefix of $o|_i$, where one of the Cases 1) or 2) holds for t.

However, if Case 1) holds for t, then s and r must generate the same word $m|_i$ by going through loops differently and thus, we must have $W_s \neq W_r$ while $m|_i \in W_s \cap W_r$. But this contradicts the assumption that T_i is non-redundant.

On the other hand, if Case 2) holds for t, then assuming that s' is the path in T_i that generates $n'|_i$, since $\forall k < u \leq k'$, $\nu_u \in \{\nu_0, \nu_1, \cdots, \nu_{k-1}, \nu_k\}$, s can be derived from s'. As a result, $m|_i$ will be derived from s'. However, since $n|_i \neq n'|_i$, s' (which generates $n'|_i$) and r (which generates $n|_i$) are not the same path in T_i. Therefore, we have $W_{s'} \neq W_r$ while $m|_i \in W_{s'} \cap W_r$. But this contradicts the assumption that T_i is non-redundant. Thus, we must have $m \notin prefix(L(Spec))$, which means that m is an implied pMSC for $Spec$.

Now, let's assume that B' is not safely realizable because of an implied scenario m that is a prefix of a member of B'. Then, two cases are possible.

First, is where $m \notin L(B')$, $m \in prefix(L(B'))$, and $\forall i \in P$, $m|_i$ is an acceptable execution of T_i. Now, if $m \in L(Spec)$, then for $Spec = (h,\ M)$, m must be the result of an acceptable execution $s = \nu_0\nu_1 \cdots \nu_k$ of h for which $m = \mu(\nu_0)\mu(\nu_1) \cdots \mu(\nu_k)$, and s must terminate at a terminal node of h (because s is a finite acceptable execution of h). Furthermore, s does not go consecutively through any loop, otherwise at least one $m|_i$ will not be a basic execution of T_i. But, since s ends at a terminal node, according to Definition 20, it must be a basic execution of h and therefore, its corresponding MSC m must be a member of B' ($m \in L(B')$). This contradicts the assumption that $m \notin L(B')$. Thus, we must have that $m \notin L(Spec)$ and m would be an implied MSC for $Spec$.

Second, is where $\exists i \in P$ such that $m|_i$ is not an acceptable execution of T_i. We show that such an implied scenario will not happen for B'. To see this, assume that there is such an implied scenario m for B' i.e. $\forall i \in P$, $\exists n \in B'$ such that $m|_i = n|_i$, $m \notin L(B')$, $m \in prefix(L(B'))$ and $\exists i \in P$ such that $m|_i$ is not an acceptable execution of T_i. Then, since $B' = B \cup X$, neither $m \in L(B)$ nor $m \in L(X)$. However, since $L(X) \in prefix(L(B))$ and also for any o in X and $i \in P$, there exists an MSC n in B such that $o|_i = n|_i$, m would be an MSC that $\forall i \in P$, $\exists n \in B$ such that $m|_i = n|_i$, $m \notin L(B)$, $m \in prefix(L(B))$ and $\exists i \in P$ such that $m|_i$ is not an acceptable execution

of T_i. But then, according to Lemma 6, m must be a member of X ($m \in L(X)$). This contradicts the previous result that says m is not a member of $L(X)$. Thus, an implied scenario such as m will not happen for B'.

What we have so far proved when B' has implied pMSCs can be summarized as:

- Implied pMSCs of B' that are not prefixes of its members are implied pMSCs for *Spec*
- Implied MSCs of B' that are prefixes of its members are implied MSCs of *Spec*

Thus, any implied pMSC for B' would be an implied pMSC for *Spec* and the theorem is proved. $\perp$

Now, we restrict the hMSCs to the bounded ones and establish the equivalence between safe realizability of $Spec = (h,\ M)$ and the union of B and X.

Theorem 2 *For Spec* $= (h,\ M)$, $h = (V, \rightarrow, \nu_0, V_t, \mu)$, *let* T_i *be a local LTS obtained by Algorithm 1. Then, if* h *is bounded and* $\forall i \in P$, T_i *is non-redundant, Spec is safely realizable iff* $B \cup X$ *is safely realizable.*

Proof:

"Only if direction"

For the proof of the only if direction we must assume that *Spec* is safely realizable and prove that $B' = B \cup X$ is also safely realizable.

Assume that *Spec* is safely realizable but B' is not. Then, B' should have an implied pMSC m, which by Theorem 1 would be an implied pMSC for *Spec*. However, this contradicts the assumption that *Spec* is safely realizable. Thus, B' must be safely realizable and the only if direction is proved.

“If direction”

For the proof of the if direction we must assume that B' is safely realizable and prove that *Spec* will be safely realizable. Instead, we show that if *Spec* is not safely realizable (*Spec* has implied pMSCs), B' will not be safely realizable (B' must have implied pMSCs).

Assume that *Spec* has an implied pMSC m. Let s be the path in $T = \prod_{i \in P} T_i$ that generates m. Moreover, let $\forall i \in P$, S_i is a set that $\forall s_i \in S_i$, s_i is a path in T_i that can generate $m|_i$. We consider two possibilities for m: a) $\forall i \in P$, $\exists s_i \in S_i$ such that s_i is a prefix of a basic path of T_i; b) $\exists i \in P$ for which any member s_i of S_i is not a prefix of a basic path of T_i.

a) First, we consider an implied pMSC m of *Spec* that is a prefix of a member of *Spec* ($m \in prefix(L(Spec))$). In this case, m must be an implied MSC (implied scenario) for *Spec* ($m \notin L(Spec)$) for which s ends up at an accepting state of $T = \prod_{i \in P} T_i$. Then, $\forall i \in P$, $\exists s_i \in S_i$ such that s_i can generate $m|_i$ and ends up at an accepting state of T_i. Thus, $m|_i$ would be a basic execution for T_i and therefore, by Lemma 4 (see the proof), $\forall i \in P$, $\exists n \in B'$ such that $m|_i = n|_i$.

Furthermore, m cannot be a member of $B' = B \cup X$ because $\forall o \in X$, $\exists i \in P$ such that $o|_i$ cannot be generated by a path in T_i that ends up at an accepting state of T_i. Thus, $m \notin X$. Also, if m is a member of B, then there must exist a basic execution t of h that ends at a terminal node and results in m. But then, since t is an execution of h that terminates at a terminal node, it would be an acceptable execution of h and thus, its corresponding MSC m must be a member of $L(Spec)$, which is not possible as we assumed that $m \notin L(Spec)$. Therefore, $m \notin B'$ and $\forall i \in P$, $\exists n \in B'$ such that $m|_i = n|_i$. This means that m is an implied MSC for B'.

Now, let's assume that m is an implied pMSC for *Spec* for which $m \notin prefix(L(Spec))$. Since $\forall i \in P$, $\exists s_i \in S_i$ such that s_i is a prefix of a basic path of T_i, by Lemma 4, $\forall i \in P$,

$\exists n \in B'$ such that $m|_i$ is a prefix of $n|_i$. But, since m is such an implied pMSC for *Spec* for which $m \notin prefix(L(Spec))$ and because $prefix(L(B')) \subseteq prefix(L(Spec))$, we must have $m \notin prefix(L(B'))$. This means that m is an implied pMSC for B'.

b) Now, assume that $\exists i \in P$ such that for any member s_i of S_i, s_i is not a prefix of a basic path of T_i.

Then, either the implied pMSC m is a prefix of a word in $L(Spec)$ or not. If $m \in prefix(L(Spec))$, then m must be an implied MSC ($m \in L(T)$ and $m \notin L(Spec)$) that ends at an accepting state of T. Let s' be a path that its only (possible) difference with s is that s' does not go consecutively through any loop. Then, if s' generates m', we must have $m' \in prefix(L(Spec))$ and $m' \notin L(Spec)$. This is because the projections of m and m' on local LTSs T_i are the same except for the number of traversing some loops. Thus, m and m' must be created by two executions t and t' of h such that t and t' are the same except that t' does not go consecutively through any loop while t might go. Thus, if $m' \in L(Spec)$, we must also have $m \in L(Spec)$, which contradicts the assumption that $m \notin L(Spec)$. Now, since t' (that creates m') does not go consecutively through any loop, and since m' ends at an accepting state of T, $\forall i \in P$, $m'|_i$ would be a basic execution of T_i that ends at an accepting state. Thus, by Lemma 4 (see the proof), $\forall i \in P$, $\exists n \in B'$ such that $m'|_i = n|_i$. On the other hand, we must have $m' \notin L(B')$ since otherwise m' would be a member of $L(Spec)$, which is not possible as we just showed that $m' \notin L(Spec)$. This means that m' is an MSC that $\forall i \in P$, $\exists n \in B'$ such that $m'|_i = n|_i$ and $m' \notin L(B')$, which means that m' is an implied MSC for B'.

Now, consider an implied pMSC m that is not a prefix of members of *Spec* ($m \notin prefix(L(Spec))$). If $s = q_0\omega_0q_1\omega_1 \cdots q_l\omega_l \cdots$ is the path in T that generates m, there must exist two states q_{k-1} and q_k in s such that: i) $s' = q_0\omega_0q_1\omega_1q_{k-2} \cdots \omega_{k-2}q_{k-1}$ does not go consecutively through any loop and generates a word m' (over the al-

phabet Σ) that is a prefix of a basic execution of h i.e. $m' \in prefix(L(B'))$; and ii) $s'' = q_0\omega_0 q_1\omega_1 \cdots q_{k-1}\omega_{k-1}q_k$ generates an execution m'' of T that is not a prefix of any word of $L(Spec)$ (m'' is an implied pMSC for $Spec$). We show that m'' must be also an implied pMSC for B'.

First, note that if $\forall i \in P$, $\exists s''_i \in S''_i$ such that s''_i is a prefix of a basic path of T_i, then we have a situation like Case a) and thus, m'' must be an implied pMSC for B'. Therefore, let assume that for s'' there exists a process $j \in P$ for which any member s''_j of S''_j is not a prefix of a basic path of T_j (which means that $m''|_j$ is not a prefix of any basic execution of T_j)

Let $\omega_{k-1} \in \Sigma_i$ for some $i \in P$. We show that with bounded hMSCs, $\exists n \in B'$ such that $m''|_i$ is a prefix of $n|_i$.

Let's assume that $\forall n \in B'$, $m''|_i$ is not a prefix of $n|_i$. Then, by Lemma 4, $m''|_i$ is not a prefix of any basic execution of T_i. Now, assuming that $s'_i = q_{i0}\omega'_0 q_{i1}\omega'_1 \cdots \omega'_{u-1}q_{iu}$ and $s''_i = q_{i0}\omega'_0 q_{i1}\omega'_1 \cdots \omega'_{u-1}q_{iu}\omega_{k-1}q_{i(u+1)}$ are the paths in T_i that generate respectively $m'|_i$ and $m''|_i$, $q_{iu}\omega_{k-1}q_{i(u+1)}$ must be a sequence in s'_i. This is because s'_i does not go consecutively through any loop and if $q_{iu}\omega_{k-1}q_{i(u+1)}$ is not a sequence in s'_i, s''_i must be a prefix of a basic path of T_i, which contradicts the assumption that $m''|_i$ is not a prefix of any basic execution of T_i. Thus, $q_{iu}\omega_{k-1}q_{i(u+1)}$ must be a sequence in s'_i and by appending $\omega_{k-1}q_{i(u+1)}$ to s'_i, s''_i must be going through a loop c_i that s'_i has already gone through it.

Now, since s'_i goes through the loop c_i, s' must also go through a loop c that its projection on T_i is c_i, otherwise T_i would be redundant. Also, since q_{iu} is a state in the loop c_i and also is the local state of process i in the state q_{k-1} of the product LTS $T = \prod_{i \in P} T_i$, q_{k-1} must also be a state in the loop c. Thus, since $q_{iu}\omega_{k-1}$ is a sequence in the loop c_i in s'_i, $q_{k-1}\omega_{k-1}$ is a sequence in the loop c in s'.

On the other hand, since h is bounded, there is a kind of synchronization between processes such that $\forall i \in P$, s'_i cannot finish the loop c_i unless any other process $j \neq i$

for which there exists a path s'_j that has a loop c_j that communicates messages with c_i, also has finished c_j. In other words, when s'_i goes through c_i once, $\forall j \in P$, $j \neq i$, if a path s'_j has a loop c_j that communicates messages with c_i, s'_j also goes through c_j once. Consequently, when s'_i finishes the loop c_i for the first time and reaches q_{iu}, s' must also finish the loop c and reaches q_{k-1}.

Now, consider q_{k-1} as the starting point of the loop c and assume that the state after $q_{k-1}\omega_{k-1}$ in s' is q'_k i.e. $q_{k-1}\omega_{k-1}q'_k$ is a sequence in s'. Then, according to the definition for the states of the product LTS T (see Section 5.4.2), q'_k is comprised of the local states of all processes $j \neq i$, which are the same as their local states in q_{k-1}, the local state of process i, which is $q_{i(u+1)}$, and the contents of the buffers between processes, which are the same as the buffers in q_{k-1} except for process i and another process j that i communicates with by ω_{k-1} for which their buffers are completely determined by ω_{k-1}.

But, if we compare two sequences $q_{k-1}\omega_{k-1}q_k$ and $q_{k-1}\omega_{k-1}q'_k$, all the local states and buffers of processes as well as the transition ω_{k-1} are the same for q'_k and q_k. As a result, we must have $q'_k = q_k$. This means that $q_{k-1}\omega_{k-1}q_k$ is a sequence in s'. But, this means that s'' is going through the same loop c that s' has already passed and as a result, m'' must also be a prefix of $L(Spec)$, which contradicts the fact that m'' is an implied pMSC for $Spec$.

Thus, $m''|_i$ must be a prefix of $n|_i$ for some $n \in B'$. On the other hand, since also $\forall j \in P$, $j \neq i$, $m'|_j = m''|_j$, we have that $\forall i \in P$, $\exists n \in B'$ such that $m''|_i$ is a prefix of $n|_i$. Moreover, since $m'' \notin prefix(L(Spec))$ and $prefix(L(B')) \subseteq prefix(L(Spec))$, we must have $m'' \notin prefix(L(B'))$. This means that m'' is an implied pMSC for B' and completes the proof for the if direction. Thus, the theorem is proved. $\perp$

We should notice that since basic executions are the result of the choice node properties of hMSCs, Theorem 2 shows that implementation problems of MSC specifications

are the result of an effect of choice nodes that is captured in the set of basic executions. On the other hand, since Proposition 1 relates implied pMSCs of a set of MSCs like $B \cup X$ to the indeterministic behaviour of processes, we can conclude that indeterministic behaviour of processes is the main problematic property that is developed by choice nodes, which together with the set $B \cup X$ can explain why between two MSC specifications one is safely realizable while the other is not.

5.5 Algorithm

Using Theorem 2, in order to check safe realizability of *Spec*, safe realizability of $B \cup X$ can be checked. However, in practice we can check safe realizability for B while ignoring implied scenarios of B that are members of X. These implied scenarios can be calculated separately, for instance, using an algorithm that is given in [21].

Algorithm 2 shows how safe realizability for MSC specifications is reduced to safe realizability for a set of MSCs.

Algorithm 2: Checking safe realizability for $Spec = (h, M)$ (h is bounded and $\forall i \in P$, T_i is non-redundant).

1. Compute all basic executions of h
2. For each basic execution obtained in Step 1 construct an MSC that corresponds to the concatenation of the MSCs of its nodes to obtain a set B of finite MSCs
3. Check whether there exists any implied pMSC for B while ignoring implied pMSCs that are members of X. If there exist no implied pMSCs for B, *output Spec* is safely realizable; otherwise, *output Spec* is not safely realizable

Completeness and correctness of the algorithm is provided by Theorem 2. Note that, by Proposition 1, in the last step the algorithm checks whether indeterministic behaviour of processes in members of B results in such implied pMSCs for B that are not members of X. To give the exact cause of implied pMSCs, the algorithm can be modified to show the implied pMSC, the path in the hMSC that leads to that implied pMSC, and the process and its related indeterministic behaviour that cause such an implied pMSC. This information is useful when we want to correct the specification in such a way that the detected implied pMSC will not happen any more.

In Chapter 7, an example for the application of Algorithm 2 will be illustrated.

5.5.1 Complexity of the Algorithm

In the following, we put a worst case upper bound of $O(K^5 2^{4K^2+5K} K' K''^2)$ on the complexity of Algorithm 2 where for $Spec = (h, M)$, K is the number of nodes in h, K' is the number of events in the set of MSCs M, and $K'' = |P|$ is the number of processes. This result is consistent with the result that safe realizability for bounded hMSCs is EXPSPACE-complete [24, 26].

We start with calculating basic executions of h. If we number the nodes in h from 0 to $K-1$ and by R^k_{0j} we denote a path in h between nodes 0 and j that does not go through nodes greater than k, then we are interested in R^K_{0j} when j varies from 1 to $K-1$. However, constructing R^K_{0j} when j varies from 1 to $K-1$ has the same complexity as the complexity of converting an automaton to a regular expression. Thus, assuming that an hMSC has K nodes, this step can be done in time $O(K^3 4^K)$ [20]. Also, since there exist no more than $K(K+1)$ edges in h, the total number of basic executions in each R^K_{0j} will not be more than the number of possible subsets of these K^2+K edges or $2^{K^2+K}-1$. Therefore, the total number of basic executions in h (for K nodes i.e. when j varies from 1 to $K-1$) will not be more than $K2^{K^2+K}$.

Also, a basic execution in each R_{0j}^{K} with the maximum length is the one that goes through all the loops exactly once. For $2^K - 1$ possible loops, and K as the maximum number of nodes in a loop, the length of a basic execution will not be more than $K2^K$. Now, because loops of basic executions are already detected in the process of construction of each R_{0j}^{K}, finding those executions that satisfy Criteria i), ii), and iii) of Definition 20 can be done in time $O(K^4 2^{K^2+3K})$ ($K2^{K^2+K}$ multiplied by $(K2^K)^2$) and comparing them to find whether one is a prefix of another (for Criterion iv) of Definition 20) can be done in time $O(K^3 2^{2K^2+3k})$ ($(K2^{K^2+K})^2$ multiplied by $K2^K$). Therefore, Step 1 can be done in time $O(K^3 4^K + K^4 2^{K^2+3K} + K^3 2^{2K^2+3K})$ or $O(K^4 2^{2K^2+3K})$.

On the other hand, the corresponding MSC of a basic execution of h can be obtained in the order of the length of a basic execution ($K2^K$) multiplied by the number of processes K'' and the number of events K'. Therefore, for all basic executions ($K2^{K^2+K}$), Step 2 can be done in time $O(K^2 2^{K^2+2K} K' K'')$.

Furthermore, for the set of MSCs B with $|B|$ MSCs, K' events in all MSCs, and K'' processes, based on Proposition 1, checking for implied pMSCs for B can be done in time $O(|B|^3 K' K''^2)$. However, first in our case $|B|$ is not more than $K2^{K^2+K}$. Second, if the number of events in the set of MSCs M is K', because of the concatenation operation that is performed in order to obtain the corresponding MSCs of basic executions of h, the total number of events in the set B will be multiplied by a factor that will not be more than the number of nodes in a basic execution ($K2^K$) multiplied by the number of basic executions ($K2^{K^2+K}$). As a result, Step 3 can be done in time $O(K^5 2^{4K^2+5K} K' K''^2)$. Thus, the whole algorithm (Step 1, Step 2, and Step 3) takes no more than $O(K^4 2^{2K^2+3K} + K^2 2^{K^2+2K} K' K'' + K^5 2^{4K^2+5K} K' K''^2)$ or $O(K^5 2^{4K^2+5K} K' K''^2)$ time.

5.5.2 The Bottleneck of Complexity

The complexity of the algorithm seems disappointing for practical purposes considering current computer hardware technology. However, this complexity should be considered by taking account of the following three factors.

At first, we should notice that the source of this complexity is the factor of 2^{K^2+K} that is obtained based on a worst case assumption where there is a pair of edges between each pair of nodes in an hMSC. While such an hMSC graph can be imagined in theory, it is unlikely to happen in practice for engineering systems. For instance, for the 4 nodes hMSC of the example of the Boiler Control system given in Chapter 7, the number of edges is 5 while in the worst case it could have 20 edges. Considering the fact that for a given hMSC the number of edges will affect the computation time as powers of 2, there would be a significant difference between the run time of the worst case and the run time of a realistic case.

Secondly, we have only provided a basic algorithm that checks for safe realizability in asynchronous message setting. Due to the scope of this thesis, we do not investigate the issue related to how to improve the complexity of the computation and leave this as a candidate direction for future work. For instance, optimization techniques in the graph theory such as those that are used for finding shortest paths and cycles might be effective in finding basic executions of hMSCs (as directed graphs) and reducing the complexity of the algorithm (which can improve the speed). In [21] for instance, a graph theory cycle detection technique is utilized to reduce the complexity of a polynomial time algorithm.

Furthermore, if finding deadlocks and implied scenarios can be properly expressed as an optimization problem, then as an alternative approach, artificial intelligence (AI) based techniques (such as genetic algorithms) might also be promising for checking safe realizability. While such alternative approaches might run faster than our basic algorithm, nevertheless, since they search through (randomly) selected parts in the possible

search space of the problem, they will sacrifice their completeness in order to find the solution faster. This means that those AI-based approaches cannot guarantee to find all possible deadlocks and implied scenarios for a given specification. In contrast, our basic algorithm aims at finding all possible deadlocks and implied scenarios for a given specification. This is the scope of the thesis and we leave the optimization to the future for extending this work. Therefore, we did not explore the complexity of our algorithm in Chapter 7 for a practical example.

5.6 Summary

A number of researchers have studied implied scenarios and devised some algorithms for detecting them. However, those works are either restricted by certain assumptions such as synchronous message passing in the system [80], or their completeness/correctness (an algorithm for a problem is correct if any output of the algorithm is a solution for the problem. The algorithm is complete if any solution for the problem is an output of the algorithm) has not been proved [27]. Also, they do not answer why given two specifications, one is safely realizable while the other is not [86, 27]. Apart from its importance from a theoretical point of view, this question will be more important from a practical point of view when it comes to correct MSC specifications to prevent from anomalies such as deadlocks.

The problem underlying the notion of realizability relates also to other works on pathological choice nodes in high-level Message Sequence Charts (hMSCs), which resulted in recognizing some choice node properties such as non-local choice ([11, 27]), indeterministic choice, and race choice ([86]) as the potential problematic choice nodes. However, while the absence of these properties ensures safe realizability for choice constructs, their presence does not imply anything about realizability of MSC specifications.

Furthermore, because these properties are obtained by strengthening of the real cause of implied scenarios and deadlocks [87], they fail to provide the exact point where deviations from intended behaviours happens in the specifications.

While there is a noticeable body of research on realizability of MSCs and to a lesser extend choice nodes properties, there is no criterion that is both practical and complete for checking safe realizability of MSC specification. In this regard, our work contributes to the completeness of the realizability criterion as looking for implementation problems in the choice node properties do not result in a complete solution [11, 86, 87, 27].

Furthermore, we showed that individual choice node properties result in a more general property for the behaviour of processes that is defined as indeterministic behaviour of processes. The advantage of this property is that it is defined in terms of local behaviour of processes and can be checked using a syntax checker for MSCs.

Finally, we should notice that because our approach gives both the process and the location in its communication history for which indeterministic behaviour happens, it provides useful information for correcting specifications. This advantage and its implication in software engineering practice will be illustrated in Chapter 7.

Chapter 6

STRONG SAFE REALIZABILITY AND EMERGENT SCENARIOS

In this chapter, we study the notion of safe realizability for MSC specifications in terms of some closure conditions defined for the language of MSCs and extend it to a more general realizability notion called *strong* safe realizability. Furthermore, instead of deadlock states for MSC specifications, we introduce *stuck* states and use emergent scenarios to name both implied scenarios and those anomalies of MSC specifications that are solely captured by the notion of strong safe realizability.

6.1 Introduction

As it is said in Chapter 5, safe realizability is a measure that can tell whether there exists a deadlock free distributed implementation for an MSC specification such that it exactly covers the behaviours described by the specification. In this chapter, we scrutinize the basic assumptions behind safe realizability and show that by extending those assumptions, we can derive a stronger notion of realizability for MSC specifications.

One of the fundamental assumptions for safe realizability is that for the accepted executions of a system, the semantics of MSC specifications is interpreted as the language accepted by an automata model. Moreover, deadlocks for MSC specifications are interpreted the same as deadlocks in automata models [21]. According to this view, for a multi-process distributed system, a given execution of the system will not result in deadlocks as long as the corresponding path of such an execution in the automata model of the system can end (possibly by extending the path) at an accepting state. However,

such a view ignores whether the behaviour of individual processes in the system allows for extending a given path to an accepting state.

For instance, consider a given execution of the system in which a process reaches at an accepting state of its automata model and there is only one outgoing transition of that accepting state labeled with an action of the process. Now, if all other processes are waiting for this action to be fired and if the process does not initiate the action, the execution cannot continue for the system even though in the automata model of the system the execution can be extended to an accepting state. Thus, while assuming the same semantics for MSC specifications and automata models addresses automata-based deadlocks for a system, it will not cover all possible deadlocks for the system.

Therefore, one of the main goals of this chapter is to distinguish between the concept of deadlocks in MSC specifications and deadlocks in automata models. In so doing, instead of deadlock states, stuck states is defined for the system automata model in order to provide a consistent semantics for deadlocks in MSC specifications and automata models. Furthermore, safe realizability is reformalized based on the definition of stuck states, which results in a new notion for realizability that captures those anomalies of MSC specifications that cannot be addressed by safe realizability. The new notion of realizability is called strong safe realizability since whenever an MSC specification is strongly safe realizable, it will be also safely realizable. Also, to stick to the current definition for implied scenarios, for an MSC specification we use emergent scenarios to name implied scenarios that are captured by safe realizability as well as those new behaviours for the specification that are solely captured through strong safe realizability.

6.2 Background

In this section, a brief overview of representing MSCs as the words of a formal language will be presented. Furthermore, safe realizability will be defined based on some closure conditions defined over the language of MSC specifications when MSCs are treated as the words in some language. The reader can refer to [21] or [26] for more details.

To account for the definition of MSCs (Definition 1 in Chapter 3), well-formed and complete words are defined. A well-formed word captures the definition of a pMSC. It is defined as a word over the alphabet Σ such that for every receive event its corresponding send event exists in it. A complete word over the alphabet Σ is a word such that for every send event, its corresponding receive event exists in it. Therefore, a complete and well-formed word represents an MSC.

Using the concept of well-formed and complete words, implied pMSCs can be defined using the following closure conditions.

For a language L, let $prefix(L)$ denotes the set of all prefixes of the words in L.

Closure condition CC3: A language L over the alphabet Σ is said to satisfy closure condition CC3 iff for all well-formed words ω: if for each process i there is a word $\nu^i \in prefix(L)$ such that $\omega|_i = \nu^i|_i$, then ω is in $prefix(L)$.

Those words that falsify the closure of the language of the set of MSCs M under the closure condition CC3 are called implied pMSCs for the set M. A straightforward check for CC3 has exponential complexity. So, an equivalent condition, which can be checked by a polynomial time algorithm is defined [21]:

Closure condition CC3$'$: A language L over the alphabet Σ is said to satisfy closure condition CC3$'$ iff for all $\omega, \nu \in prefix(L)$ and all processes i: if $\omega|_i = \nu|_i$, and $\omega x \in prefix(L)$ and νx is well-formed for some $x \in \Sigma_i$, then νx is also in $prefix(L)$.

On the other hand, closure condition CC3 (as it is shown in [26]) cannot completely

address safe realizability. Thus, safe realizability is given as a combination of CC3 and another closure condition CC2$'$:

Closure condition CC2$'$: A language L over the alphabet Σ satisfies closure condition CC2$'$ iff for all well-formed and complete words ω over Σ such that $\omega \in prefix(L)$: if for all processes i there exists a word ν^i in L such that $\omega|_i = \nu^i|_i$, then ω is in L.

Then, it can be proved that a language L is safely realizable iff it satisfies both CC3$'$(equivalent to CC3) and CC2$'$ [26]. Also, a polynomial time algorithm is given in [21] for checking both CC3$'$ and CC2$'$.

6.3 Yet Another Non-Determinism

In this section, it will be shown that indeterministic behaviour of processes might introduce such emergent behaviours for MSC specifications that cannot be addressed through safe realizability.

Consider Fig. 6.1, which shows two MSCs for a system. Based on these two MSCs, there would be an indeterministic behaviour for process C1 since after sending message c, according to MSC2 it is able to send message e, while according to MSC1 it also can stop sending message e (see Case iii) of Definition 11 in Chapter 5). Consequently, the MSC of Fig. 6.2 is a valid scenario and can happen. According to this MSC, in the FSM models of processes C1 and C3, they would reach their final states and stop while process C2 is waiting to receive message e, which might never receive.

To see how Case iii) of Definition 11 in Chapter 5 is problematic, observe that based on the MSCs of Figure 6.1 there is no mechanism (in terms of some synchronizing messages) to force process C1 to send message e in Figure 6.2 because according to MSC1 the behaviour of process C1 is already a valid behaviour in Figure 6.2. However, because safe realizability is based on the automata definition for deadlocks, as long as there is an

execution such as MSC2 in Figure 6.1 that reaches an accepting state of the concurrent automata of C1, C2, and C3, the behaviour of process C1 in MSC1 is ignored. Nevertheless, we can see that based on MSC1 and MSC2, there exists no reason to believe that C1 is always following the pattern of MSC2.

Note that, Figure 6.2 is a prefix of MSC2, and therefore, it fulfills the closure condition CC3. On the other hand, since the projection of process C2 in Figure 6.2 neither is the same as its projection in MSC1 nor is the same as its projection in MSC2, the MSC of Figure 6.2 does not have the condition of the words specified in the closure condition CC2′ and thus, it cannot be addressed by CC2′.

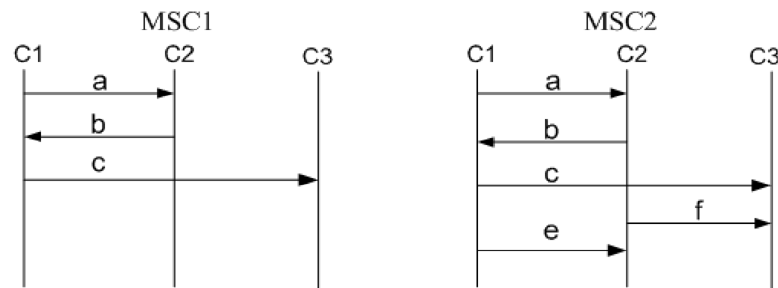


Figure 6.1: According to these two scenarios, process C1 can indeterministically decide to send or not to send message e.

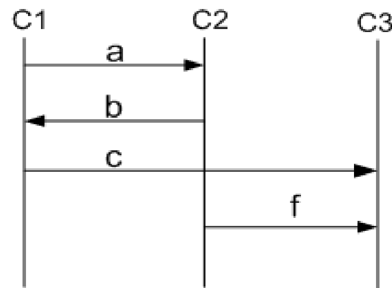


Figure 6.2: An emergent scenario from two scenarios of Fig. 6.1.

6.3.1 Strong Safe Realizability

Figures 6.1 and 6.2 show that Case iii) of Definition 11 in Chapter 5 can result in such behaviours for MSC specifications that are not addressed by the closure conditions CC3

and CC2$'$. This situation can be treated as a stronger deadlock condition than the one defined for automata models. It also resembles the notion of *stucks* in the context of Communicating Sequential Processes (CSP) [88]. Informally, stuck freeness means that a message sent by a sender will not get stuck without some receiver ever receiving it, and a receiver waiting for a message will not get stuck without some sender ever sending it.

In order to account for the situation discussed for Figures 6.1 and 6.2, we define the notion of stuck states in the context of MSC specifications and characterize it for the product automata $A = \prod_{i \in P} A_i$.

Definition 22 *(Stuck state): A reachable state q in $A = \prod_{i \in P} A_i$, is called a stuck state either if q is a deadlock state or if for every process $i \in P$ for which its local state q_i in the state q is an accepting state of A_i we remove all possible outgoing transitions $i!j(c)$ from q (for some $j \in P$ and $c \in C$), none of the accepting states of A is reachable from q.*

Then, because of the notion of stuck states (that also covers the definition of deadlock states), we reformalize safe realizability of a set of MSCs in terms of Definition 22. In order to do that, we need another closure condition called CC$_{ss}$ that addresses both the objectives of CC2$'$ together with the stronger deadlock requirements given by Definition 22.

Closure condition CC$_{ss}$: A language L over the alphabet Σ is closed under closure condition CC$_{ss}$ iff for any well-formed word $\omega \in prefix(L)$ for which $\exists P1 \subseteq P$ such that $\forall i \in P1$, $\omega|_i = \nu^i|_i$ for some $\nu^i \in L$: $\exists x \in L$ such that $\omega \in prefix(x)$ and $\forall i \in P1$, either $x|_i = \omega|_i$ or $x|_i[||\omega|_i||]$ is a receive event.

Note that, CC$_{ss}$ is a stronger closure condition than CC2$'$. This fact is represented by the following lemma.

Lemma 7 *A language L over the alphabet Σ is closed under $CC2'$ if it is closed under CC_{ss}.*

Proof:

Suppose that L satisfies CC_{ss} and consider a well-formed and complete word ω over Σ with the condition described in the closure condition $\mathrm{CC2}'$, that is $\omega \in prefix(L)$ and for all processes i there exists a word ν^i in L such that $\omega|_i = \nu^i|_i$. We must show that $\omega \in L$. Since L satisfies CC_{ss}, there exists an x in L such that ω is a prefix of x and $\forall i \in P$, either $x|_i = \omega|_i$ or $x|_i[\|\omega|_i\|]$ is a receive event. This means that there is not any send event in x in addition to the current send events in ω, and since x is a well-formed and complete word, it cannot have any receive event in addition to ω. In other words, the send and the receive events of x and ω are the same. Thus, $x = \omega$ or equivalently $\omega \in L$ and the lemma is proved. $\perp$

Now, we define the notion of strong safe realizability for a set of MSCs M and then relates it to the closure conditions CC3 and CC_{ss} by Theorem 3.

Definition 23 *A set of MSCs M is said to be Strongly Safe Realizable (SSR) iff there exists a concurrent automata $A = \prod_{i \in P} A_i$ such that A is stuck free and $L(M) = L(A)$.*

Theorem 3 *A set of MSCs M is SSR iff it is closed under both CC3 and CC_{ss}.*

Proof:

We use the results in [21], which proves that safe realizability for the language $L(M)$ corresponds to the closure of $L(M)$ under both CC3 and $\mathrm{CC2}'$.

"Only if direction"

For the proof of the only if direction we must assume that $L(M)$ is SSR i.e. there exists a stuck free product automata $A = \prod_{i \in P} A_i$ such that $L(M) = L(A)$, and show

that $L(M)$ is closed under CC3 and CC$_{ss}$. Because A is stuck free and stuck freeness implies deadlock freeness, A is deadlock free and thus from the proof given in [21], $L(M)$ must satisfy CC3 (and CC2$'$). It only remains to show that $L(M)$ satisfies CC$_{ss}$. To show this, we must show that for a well-formed word $\omega \in prefix(L(M))$ over Σ for which $\exists P1 \subseteq P$ such that $\forall i \in P1$, $\omega|_i = \nu^i|_i$ for some $\nu^i \in L(M)$: $\exists x \in L$ such that $\omega \in prefix(x)$ and $\forall i \in P1$, either $x|_i = \omega|_i$ or $x|_i[\|\omega|_i\|]$ is a receive event.

Since ω is a prefix of $L(M)$ and $\forall i \in P1$, $\omega|_i = \nu^i|_i$ for some $\nu^i \in L(M)$, by executing ω the product automata A will be in a state q in which $\forall i \in P1$, the local state of process i is an accepting state of the local automata A_i. Now, since $A = \prod_{i \in P} A_i$ is stuck free in the sense of Definition 22, if we remove the outgoing transitions of q that are labeled with the send events $i!j(c)$, $j \in P$, $c \in C$, still we can reach an accepting state of A. This means that there exists a word $x \in L(A) = L(M)$ such that ω is a prefix of x and $\forall i \in P1$, either $x|_i = \omega|_i$ or $x|_i[\|\omega|_i\|]$ is a receive event. Therefore, $L(M)$ must be closed under CC$_{ss}$.

"If direction"

For the proof of the if direction we must show that if $L(M)$ is closed under CC3 and CC$_{ss}$, then $L(M)$ is SSR. From [21], if $L(M)$ satisfies CC3 and CC2$'$, it is deadlock free and also $L(M) = L(A)$. Also, from Lemma 7, CC$_{ss}$ implies CC2$'$. Consequently, since $L(M)$ satisfies CC3 and CC$_{ss}$, we can conclude that $L(M) = L(A)$ and that A is deadlock free. It only remains to show that A is also stuck free i.e. if for any process $i \in P$ that its local state q_i in the state q is an accepting state of A_i, we remove all outgoing transitions from q that are labeled with send events $i!j(c)$ for some $j \in P$ and $c \in C$, still an accepting state of A is reachable from q.

Let ω be a word that brings A to the state q and $\exists P1 \subseteq P$ such that $\forall i \in P1$, the local states of process i in q is an accepting state of A_i. Then, $\forall i \in P1$, we have that $\omega|_i = \nu^i|_i$ for some $\nu^i \in L(M)$. Also, since $L(M)$ satisfies CC3, $\omega \in prefix(L(M))$.

Thus, by CC_{ss} there is a word $x \in L(M) = L(A)$ such that ω is a prefix of x and $\forall i \in P1$, either $x|_i = \omega|_i$ or $x|_i[\|\omega|_i\|]$ is a receive event. Since $L(M) = L(A)$, x will be an acceptable execution of A that is a continuation of ω, goes through q, and $\forall i \in P1$, it does not include any outgoing transition for q that is labeled with send events $i!j(c)$, $j \in P$, $c \in C$. As a result, if we remove all outgoing transitions $i!j(c)$ from q, still an accepting states of A will be reachable by x. Thus, the product $A = \prod_{i \in P} A_i$ will be stuck free and this completes the proof. $\perp$

Note that, since by Lemma 7, CC_{ss} implies $\text{CC2}'$, whenever an MSC specification is strongly safe realizable, it would also be safely realizable.

In [21], those words that falsify the closure of language L under closure conditions CC3 or $\text{CC2}'$ are called implied pMSCs. Because we have changed the closure condition $\text{CC2}'$ to CC_{ss}, we call those words that falsify the closure of language L under closure conditions CC3 or CC_{ss}, emergent pMSCs. In this way, emergent pMSCs for an MSC specification include implied pMSCs together with those executions that result in stuck states in all the product automata that are constructed from the specification. Formally, we can define emergent pMSCs as follows:

Definition 24 *Emergent pMSCs. A word ω over the alphabet Σ is an emergent pMSC for a set of MSCs M either if ω is an implied pMSC for M or if $\omega \in prefix(L(M))$, $\exists P1 \subseteq P$ such that $\forall i \in P1$, $\omega|_i = \nu^i|_i$ for some $\nu^i \in M$, but $\forall x \in M$ for which $\omega \in prefix(x)$, $\exists i \in P1$ such that $x|_i[\|\omega|_i\|]$ is a send event.*

6.4 Summary

In this chapter, we extended the notions of deadlock states and safe realizability, respectively to stuck states and strong safe realizability. Also, we showed that the extended

realizability notion (strong safe realizability) captures such anomalies for MSC specifications that are not covered by its previous counterpart (safe realizability).

Chapter 7

APPLICATION EXAMPLE

In this chapter, the definitions and the algorithm presented in Chapter 5 are illustrated using a boiler control system example consisting of four processes: Control, Sensor, Database, and Actuator. The control unit operates the sensor and the actuator to control the pressure of a steam boiler. The database is used as a repository to buffer the sensor information while the control unit performs calculations and sends commands to the actuator. The specification for the system can be viewed in Figure 7.1.

7.1 Detecting Emergent Scenarios

To illustrate Algorithm 2 in Chapter 5, consider the MSC specification of Figure 7.1 for a Boiler Control system. This figure differs from the specification given in [27] in that there is no self loop on the Register node in the hMSC in order to provide boundedness property for it as safe realizability for unbounded hMSCs is undecidable [26].

Furthermore, it should be noted that in [27] and [80], the Boiler Control system is analyzed for detecting implied MSCs not implied pMSCs. As a result, these works do not check for safe realizability of the system. Rather, they check for a weaker notion of realizability that only considers implied MSCs and ignores deadlocks caused by implied pMSCs. Also, since when asynchronous message passing is assumed between processes, checking for implied scenarios of MSC specifications is undecidable [24] (note that checking for implied scenarios is different than checking for safe realizability), they assume that synchronous message passing holds between processes. In contrast, in this chapter we check for implied pMSCs (safe realizability) of the system and assume that asynchronous

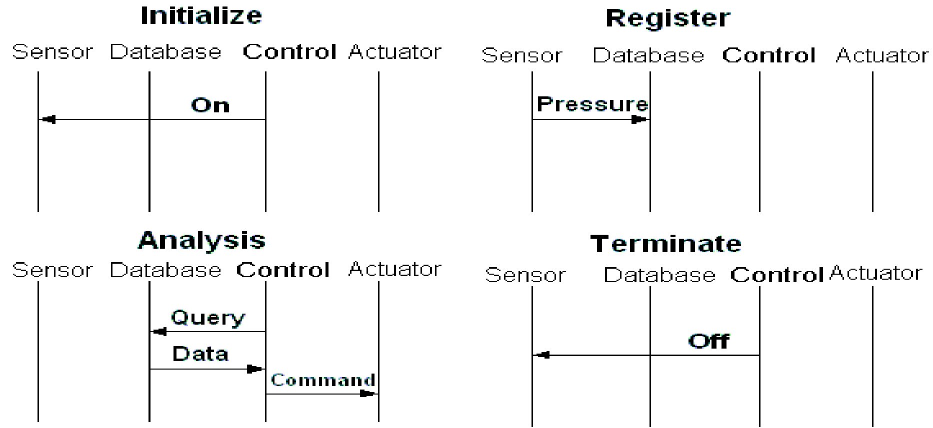
message passing holds between processes.

As it is shown in Figure 7.1, mapping between nodes and MSCs are: $\mu(\nu_0) = \mathit{Initialize}$, $\mu(\nu_1) = \mathit{Register}$, $\mu(\nu_2) = \mathit{Analysis}$, and $\mu(\nu_3) = \mathit{Terminate}$. Local LTSs for the processes of this system are also shown in Figure 7.2. As it is depicted in Figure 7.2, all LTSs are deterministic. Therefore, all the LTSs are non-redundant and thus, we can check for safe realizability using Algorithm 2.

Basic executions of the hMSC of Figure 7.1 are $s_1 = \nu_0.\ \nu_1.\ \nu_3.\ \nu_0.\ \nu_1.\ \nu_2.\ \nu_1$ and $s_2 = \nu_0.\ \nu_1.\ \nu_2.\ \nu_1.\ \nu_3.\ \nu_0$ (Step 1 of Algorithm 2). Note that, a path like $s_3 = \nu_0.\ \nu_1.\ \nu_3.\ \nu_0$ also goes through a loop only once and the only node reachable after the last node (ν_0) in this path is ν_1, which is a node in s_3. Thus, it has the first and the second conditions of a basic execution for the hMSC of Figure 7.1 (see Definition 20 in Chapter 5). Also, $\nu_3.\ \nu_0$ is not a repeated sequence in s_3. Thus, it satisfies the third condition for a basic execution. However, since s_3 is a prefix of s_1, it cannot be a basic execution for the hMSC of Figure 7.1.

The corresponding MSCs resulting from concatenating of the MSCs in the basic executions s_1 and s_2 are respectively: m_1=*Initialize. Register. Terminate. Initialize. Register. Analysis. Register* and m_2=*Initialize. Register. Analysis. Register. Terminate. Initialize*, which are shown in Figure 7.3 (Step 2 of Algorithm 2). Because s_1 and s_2 are the only basic executions for the hMSC of Figure 7.1, $B = \{m_1, m_2\}$.

For the MSCs of Figure 7.3, indeterministic behaviour of Database in m_1 because of m_2 (Case ii) of Definition 11 in Chapter 5) makes it possible to replace the second *Pressure* message for this process in m_1 with the message *Query* (note that this replacement is valid according to m_2) and get the implied scenario of Figure 7.4a). Similarly, in m_2 Sensor has indeterministic behaviour because of m_1, which makes it possible to replace the second *Pressure* message for this process in m_2 with the message *Off* and get the implied scenario of Figure 7.4b) (Step 3 of Algorithm 2). Note that, the scenarios of



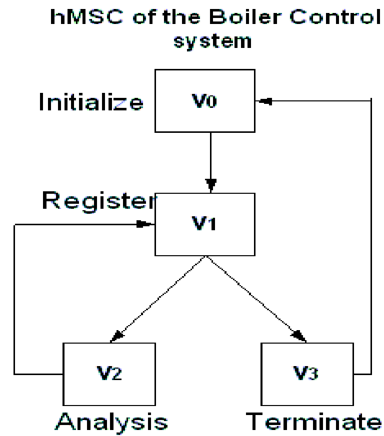


Figure 7.1: MSC specification for a Boiler Control system.

Figure 7.4 are not intended by the specification of Figure 7.1. For instance, for Figure 7.4a) while Sensor, Actuator, and Control are acting according to a prefix of the path s_1, because of its indeterministic behaviour in m_1, Database is acting according to a prefix of the path s_2. This lack of synchronization neither is intended by the hMSC of Figure 7.1 nor is a prefix of MSCs m_1 and m_2 (obtained from the basic executions of the hMSC of Figure 7.1).

Note that, for $B = \{m_1, m_2\}$, the emergent pMSCs of Figure 7.4 are not prefixes of m_1 or m_2. As a result, none of them can be a member of X (the set whose members are emergent pMSCs that are prefixes of members of B) and ignored by Algorithm 2.

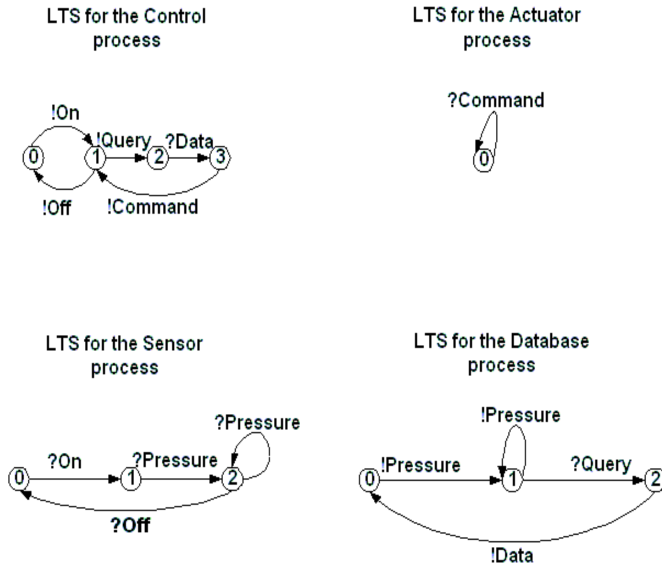


Figure 7.2: LTS models of the processes in the Boiler Control system.

Thus, the set $B' = B \cup X$ has emergent pMSCs (shown in Figure 7.4) and the algorithm outputs that the specification shown in Figure 7.1 is not safely realizable (also, the emergent pMSCs like those in Figure 7.4 can be shown).

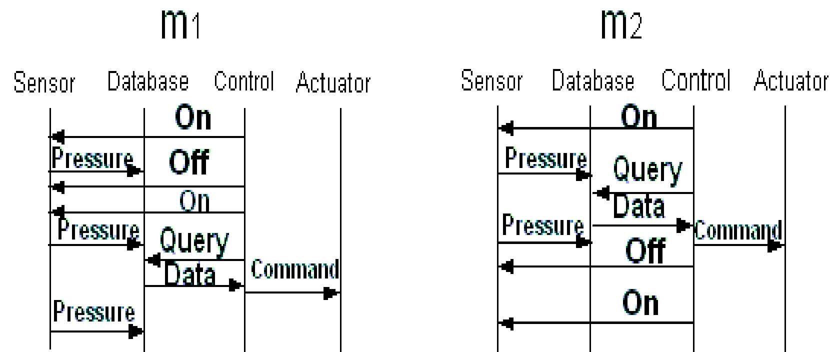


Figure 7.3: Two MSCs obtained from two basic executions of the hMSC of Figure 7.1.

7.2 Correcting MSC Specifications

To illustrate the practical implications of the results of Chapter 5, suppose that we want to correct the specification of Figure 7.1 by some synchronizing messages such that the

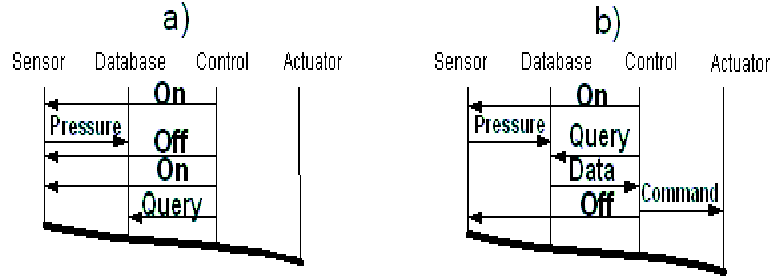


Figure 7.4: Two emergent pMSCs obtained from two MSCs of Figure 7.3.

emergent scenarios of Figure 7.4 cannot happen anymore. The question is where would be the best place to insert a synchronizing message in the specification. We explain how we can accomplish this goal using the result of Section 7.1. By referring to the MSCs of Figure 7.3, we should prevent from indeterministic behaviour for the Database process in m_1 (and the Sensor process in m_2), which means that some synchronizing messages must be used such that by removing the receiving event of the second *Pressure* message on Database (the sending event of the second *Pressure* message on Sensor), the Control process cannot send the *Query* (*Off*) message anymore (see Case ii) of Definition 11 in Chapter 5). One way to achieve this, is to let Database sends a *Sync* message to Control after receiving the *Pressure* message. This new message must be a syntactical cause (Definition 2 in Chapter 3) for the sending events of *Query* and *Off* messages on the Control process. Considering three possible scenarios of Register, Analysis, and Terminate that we can insert the new message, the best place to introduce the new *Sync* message would be right after the *Pressure* message in the MSC Register to cover both the sending event of *Query* in the MSC Analysis and the sending event of *Off* in the MSC Terminate.

The new Register scenario is shown in Figure 7.5. The modified Register scenario prevents from undesired behaviours of Figure 7.4, since there would be no chance for indeterministic behaviour of the Database and the Sensor processes in the new MSCs m_1' and m_2' (Figure 7.6) resulting from basic executions of the hMSC of Figure 7.1.

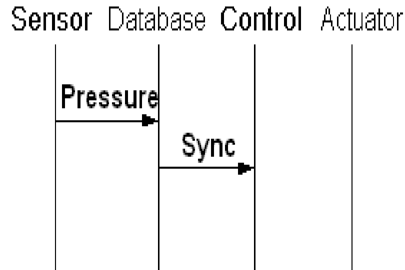


Figure 7.5: Modified Register scenario.

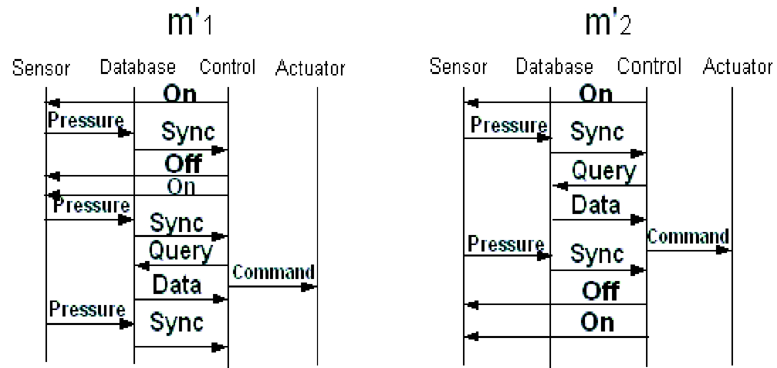


Figure 7.6: Corresponding MSCs of the basic executions s_1 and s_2 after modifying the Register MSC.

Chapter 8

SUMMARY AND OUTLOOK

There are many ways for modeling dynamic behaviour of systems. Two popular approaches are scenario-based and state-based modeling. A scenario-based model represents the inter-object behaviours by describing interactions between multiple objects or processes. In contrast, a state-based model is often used to represent the entire behaviour of a system, for instance by a set of communicating state machines in which each state machine describes an intra-object view for the behaviour of an object.

Scenarios have the advantage to give a view of the system activity such that causalities and concurrencies are explicitly represented. Nevertheless, composing scenarios is not an easy task. In this regard, transforming a scenario specification into a set of communicating state machines is a way for composing scenarios and a first step toward the design and implementation of the specification.

When synthesizing state machines from scenarios, one might expect that the synthesized state machines correctly reflect the behaviours specified by scenarios. What might happen instead is that the state-based model that is synthesized from the scenarios presents sets of behaviours that do not appear in scenarios. These unexpected behaviours are called emergent behaviours and can arise due to two separate effects. On one hand, due to a scenario-based specification and regardless of the technique used in the synthesis process, the synthesized state machine might include unexpected behaviours that are not specified by the specification. These unexpected behaviours are called emergent scenarios. On the other hand, due to the specific technique used for the synthesis process, new behaviours might arise for the system components. This is a phenomenon that is also studied as overgeneralization.

This work has resulted in a number of contributions in inferring emergent behaviours for scenario-based specifications that are summarized in Section 8.1.

8.1 Summary of Contributions

The main contributions of our work are as follows.

First, given a system that is represented by a scenario-based specification, for the purpose of synthesizing behaviour models from scenarios, the domain knowledge is represented by a light domain theory. The domain theory is based on invariant properties of software and systems, which makes it reusable amongst different domain experts.

Second, a behaviour modeling technique is devised that results in the same behaviour models when it is applied by different domain experts.

Third, a realizability model is introduced that extends safe realizability and captures more emergent behaviours and anomalies for scenario-based specifications.

Fourth, an algorithm is devised that can check for safe realizability in asynchronous setting in the presence of hMSCs.

Fifth, a question regarding the criteria under which MSC specifications can be safely (without deadlocks and implied scenarios) implemented is answered.

Sixth, emergent scenarios (including implied scenarios) and overgeneralization (emergent behaviours of components), which in the past were studied as separate phenomena are characterized as two different effects of indeterministic behaviour of processes.

8.2 Outlook

Several directions can be envisaged for extending this work.

First, our behaviour modeling technique can bridge between those works that start system development with scenario-based specifications [22, 28, 29], and those that start

with state machines requirements specifications [89]. State machine languages like RSML [19] have advantage in providing effective static and dynamic analysis for requirements specifications. They also have been successfully applied to mission critical and complex systems [90]. However, starting system development by state machine requirement specifications needs a good deal of effort and expertise from stakeholders.

On the other hand, scenario-based specifications are known for their ease of use by stakeholders. However, they are only examples that give partial stories of system's behaviour. Therefore, state machine requirements specifications encompassing high-level requirements for a system and scenario-based specifications that describe examples of how those requirements would affect system dynamics provide complementary descriptions of systems [91, 92].

In particular, since our behaviour modeling method abstracts the timing between messages away, similar to [30], it can be utilized for inferring declarative requirements specifications from scenarios. To infer these type-level requirements from instance-level scenarios, proper generalization and timing abstraction is required that is not supported by other behaviour modeling techniques.

Second, we can extend the work by enriching the final state machines with hierarchy and concurrency in order to provide more readability in behaviour models [29].

Third, our behaviour modeling technique can be applied in the reverse engineering of object-oriented software where event traces are generated as the result of running the target software under a debugger. The event traces can be represented as scenario diagrams and then used as the input of our behaviour modeling method in order to generate state machines that represent software requirements as well as the behaviour of objects.

Fourth, conducting large-scale case studies provide more evidence for evaluating the practical consequences of the techniques presented in Chapters 3 and 4.

And finally, as it is discussed in Chapter 5 (Section 5.5.2), we have only provided a basic algorithm that checks for safe realizability in asynchronous message setting. A candidate direction for extending this work is how to improve the complexity of the computation through appropriate optimization techniques.

Appendix A

MSC ANALYSIS TOOLS

In this appendix, first a brief overview of our Java implementation for the algorithm that checks for safe realizability (discussed in Chapters 5) is presented. Then, a few other tools for the analysis of MSCs will be discussed.

The basic building block of any MSC analysis tool is an MSC. In this regard, we chose to represent MSC specifications in plain text. One such a text format for describing the specification of Figure 7.1 is shown below.

msc Initialise:

Sensor: r-on-Control-0;

Database:;

Control: s-on-Sensor-0;

Actuator:;

endmsc

msc Analysis:

Sensor:;

Database:r-query-Control-0,s-data-Control-1;

Control:s-query-Database-0,r-data-Database-1,s-command-Actuator-0;

Actuator:r-command-Control;

endmsc

msc Terminate:

Sensor:r-off-Control-0;

Database:;

Control:s-off-Sensor-0;

Actuator:;

endmsc

msc Register:

Sensor:s-pressure-Database-0;

Database:r-pressure-Sensor-0;

Control:;

Actuator:;

endmsc

hmsc:

Start -> Initialise;

Initialise -> Register;

Register -> Analysis;

Register -> Terminate;

Terminate -> Initialise;

Analysis -> Register;

endhmsc

A text specification such as the one above has a simple syntax for representing MSCs and hMSCs. Within such a file, the definition of each MSC begins with the keyword *msc* and ends with *endmsc*. Within each MSC, processes are defined line by line. A message for a process is defined by its type (‘s’ for send and ‘r’ for receive), name of the message, and the corresponding process where the message is sent or received followed by the position on this process for the message, where the first position starts with ’0’. Note that, for each process including the position of a message that the process sends or receives is necessary in order to model the FIFO queues between processes.

Finally, at the end of file, the hMSC that defines how one MSC continues to other MSCs is also described.

The Java class *Msc* is used to hold individual MSCs.

```
public class Msc {
public Vector row=new Vector();
/** Creates a new instance of Msc */
public Msc() {
}
}
```

Then, assuming that the MSC specification has the name "input.text", the number of MSCs is 'N' and the number of processes is 'P', individual MSCs can be red into an array of *Msc* classes as follows:

```
FileInputStream inputFile=new FileInputStream("input.txt");
StreamTokenizer tokens=new StreamTokenizer(inputFile);
MSC=new Msc[N][P];
int n,p;
while (tokens.nextToken()!=tokens.TT_EOF){
for (n=0;n < N;n++){
for (n=0;n< N;n++){
p=0;
do {MSC[n][p]=new Msc();
int testEndLineAndFile=tokens.nextToken();
while((testEndLineAndFile!=tokens.TT_EOL)&&
(testEndLineAndFile!= tokens.TT_EOF)){
MSC[n][p].row.addElement((String) tokens.sval);
testEndLineAndFile=tokens.nextToken();
}
p++;} while(p<P);
```

```
}
}
inputFile.close();
}
```

Several other classes and methods are also defined for combining (sequential and parallel), comparing, and building MSCs. For instance, several methods are defined to see whether a process in an MSC has indeterministic behaviour or not. One such a method is defined to find the first position in a given process where two MSCs are different as follows:

calculateDifference(Msc[] MSC1, Msc[] MSC2, int process)

After searching for indeterministic behaviour of processes in basic executions, if any emergent scenario has been found it would be output in text format similar to the input specifications. For instance, after giving the input specification of Figure 7.1, we will get the following output, which is the text representation of the two implied scenarios in Figure 7.4.

```
msc 1:
Sensor:r-on-Control-0, s-pressure-Database-0, r-off-Control-1, r-on-Control-2;
Database:r-pressure-Sensor-1, r-query-Control-1;
Control:s-on-Sensor-0, s-off-Sensor-2, s-on-Sensor-3, s-query-Database-1;
Actuator:;
endmsc
msc 2:
Sensor:r-on-Control-0, s-pressure-Database-0, r-off-Control-4;
Database:r-pressure-Sensor-1, r-query-Control-1, s-data-Control-2;
Control:s-on-Sensor-0, s-query-Database-1, r-data-Database-2, s-command-Actuator-0,
s-off-Sensor-2;
```

Actuator:r-command-Control-3;

endmsc

We close this brief description of the Java implementation and give an account of some tools available for MSC analysis.

MSC + Tool

This is a Java program developed for both Windows and Unix platforms by Mesfin Belachew at the Tata Institute of Fundamental Research in which, first an hMSC is created in graphical form. Then the MSC parts of the hMSC are generated in graphical form. A simulation of the MSCs is also possible. Furthermore, verification is available such that the user gets implied MSCs.

INPUT: An MSC system specification.

OUTPUT: Implied MSCs.

LINK: http://www.tcs.tifr.res.in/ mesfin/MSCProject/

LTSA - MSC

The LTSA - MSC Analyser tool is developed in Java for Windows and Unix platforms and is an extension of the Labeled Transition System Analyzer (LTSA) that supports synthesis of behaviour models from scenario-based specifications and detection of implied scenarios. The tool reads an MSC specification and generates a Finite State Process (FSP) specification that is a minimal implementation model of the MSC specification.

INPUT: An MSC specification.

OUTPUT: A labeled transition system and implied MSCs.

LINK: http://www.doc.ic.ac.uk/ su2/Synthesis/

SanDriLa

This is a commercial tool developed by Sandrila Ltd as a plugin for Microsoft Visio. It is only intended to provide the stencils for drawing MSCs and HMSCs in Microsoft Visio.

LINK: http://www.sandrila.co.uk

MSC parser 2000

A Java tool that works with Windows and Unix and has been developed at the University of Lbeck at the Institute for Telematik. It requires ANTLR (ANother Tool for Language Recognition), which is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages.

INPUT: A textual representation of a MSC-2000 specification.

OUTPUT: A graphical overview in tree form is shown

LINK: http://www.itm.mu-luebeck.de

Cinderella SDL

This is a commercial tool that works only with Windows platform and has been developed by Cinderella ApS in Denmark. In fact, it is a general software development tool that supports standardized notations such as SDL, MSC, and UML for system design, specification, implementation, and simulation.

LINK: http://www.cinderella.dk

MESA

MESA (MSC Editor, Simulator and Analyzer) was originally developed at the University of Waterloo, Canada, and is now maintained by the tele research group of the University of Freiburg. This tool works only with Unix (Solaris). MESA will implement syntactic checks for non-local choice, process-divergence, timing consistency, various semantics analysis functions, and code synthesis.

INPUT: MSCs. Mesa contains a graphical MSC editor.

OUTPUT: Static analysis results (non-local choice, process-divergence, etc)

Link: http://tele.informatik.uni-freiburg.de/$\tilde{l}$eue/msc.html

Smyle

Smyle is an acronym for Synthesizing Models bY Learning from Examples and is developed by Carsten Kern, Benedikt Bollig, and Martin Leucker at the RWTH Aachen university, Germany. Its major objective is to synthesize automata models of concurrent systems from a set of MSCs comprising positive or negative system behaviours. Positive MSCs describe system behavior that is possible and negative MSCs characterize unwanted or forbidden behaviour.

Smyle employs a dedicated learning technique to generate system models that conform with the given positive and negative examples of the system behaviour.

INPUT: A set of positive and negative MSC examples for the system behaviour

OUTPUT: Concurrent automata models for the system

LINK: http://www.smyle-tool.org/

Bibliography

[1] A. van Lamsweerde and R. Darimont. Formal refinement patterns for goal-driven requirements elaboration. In *Fourth ACM SIGSOFT Symp. On Foundations of Software Eng.*, pages 179–190, 1996.

[2] W.N. Robinson. Integrating multiple specifications using domain goals. In *Fifth Intl Workshop on Software Specification and Design*, pages 219–225, 1989.

[3] J.M. Carroll. *Scenario-based design: envisioning work and technology in system development*. Wiley, New York, 1995.

[4] P. Harmon and M. Watson. *Understanding UML: The Developers Guide*. Morgan Kaufmann Publishers, San Fransisco, 1998.

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

[6] *Rational*. Available at . http://www.rational.com/, 2002.

[7] S. Heymer. A semantics for MSCs based on Petri net components. In *2nd Workshop on SDL and MSC (SAM'00)*, pages 262–275, 2000.

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

[9] J.P. Katoen and L. Lambert. Pomsets for Message Sequence Charts. In *1st Workshop on SDL and MSC (SAM'98)*, pages 197–208, 1998.

[10] M.A. Reniers. *Message Sequence Charts: Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, 1999.

[11] H. Ben-Abdhallah and S. Leue. Syntactic detection of process divergence and non-local choice in Message Sequence Charts. In *Third Int'l Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, pages 259–274. LNCS 1217, 1997.

[12] G.J. Holzmann, D. Peled, and M.H. Redberg. Design tools for requirement engineering. *Bell Labs Technical Journal*, 2(1):86–95, 1997.

[13] Y. Wakahara, Y. Kayuda, A. Ito, and E. Utsunomiya. Escort: An environment for specifying communication requirements. *IEEE Software*, 2(6):38–43, 1989.

[14] E.M. Clarke and J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[15] K. Yue. What Does It Mean to Say that a Specification is Complete? In *Fourth Intl Workshop Software Specification and Design,*, Monterey, 1987.

[16] A. van Lamsweerde, R. Darimont, and E. Letier. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Trans. On Sofware. Eng., special issue on Inconsistency Management in Software Development*, 24(11):908–926, 1998.

[17] *Unified Modelling Language*. Available at http://www.omg.org/, 2002.

[18] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[19] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Trans. on Software Eng.*, 20(9):684–707, 1994.

[20] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[21] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. *IEEE Trans. on Software Eng.*, 29(7):623–633, July 2003.

[22] E. Mäkinen and T. Systä. MAS - an interactive synthesizer to support behavioral modeling in UML. In *ICSE 2001*, pages 15–24, 2001.

[23] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *10th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'02)*, pages 109–118, 2002.

[24] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, Feb 2005.

[25] L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from HMSCs. In *5th Int'l Workshop on Formal Methods for Industrial Critical Systems*, pages 203–224, 2000.

[26] M. Lohrey. Safe realizability of high-level Message Sequence Charts. In *CONCUR 2002*, pages 177–192. LNCS 2421, 2002.

[27] H. Muccini. Detecting implied scenarios analyzing non-local branching choices. In *FASE 2003*, pages 372–386, 2003.

[28] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. on Software Eng.*, 29(2):99–115, 2003.

[29] J. Whittle and J. Schumann. Generating statecharts designs from scenarios. In *ICSE 2000*, pages 314–323, 2000.

[30] A. van Lamsweerde and Willemet L. Inferring declarative requirements specifications from operational scenarios. *IEEE Trans. on Software Eng.*, 24(12):1089–1114, 1998.

[31] A. Mousavi and B. Far. Eliciting scenarios from scenarios. In *20th International Conference on Software Engineering and Knowledge Engineering (SEKE 2008)*, pages 466–471, 2008.

[32] A. Mousavi and B. Far. Harnessing overgeneralization in the synthesis of state machines from scenarios. In *21th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2008)*, pages 107–112, 2008.

[33] A. Mousavi and B. Far. Revisiting safe realizability of Message Sequence Charts specifications. In *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*, pages 37–45, 2008.

[34] A. Mousavi, B. Far, and A. Eberlein. An approach for generating state machine designs from scenarios. In *11th IASTED International Conference on Software Engineering and Applications*, pages 88–93, 2007.

[35] A. Mousavi, B. Far, A. Eberlein, and B. Heidari. Strong safe realizability of Message Sequence Chart specifications. In *International Conference of Fundamentals of Software Engineering (FSEN)*, pages 334–349, 2007.

[36] A. Mousavi, B. Far, and A. Eberlein. *The Problematic Property of Choice Nodes in high-level Message Sequence Charts*. Technical report, Department of Electrical and Computer Engineering, University of Calgary, http://www.enel.ucalgary.ca/~amousavi/reports/choice_nodes.pdf, 2006.

[37] A. Mousavi, , A. Eberlein, and J. Denzinger. Capturing non-determinism in the system specification for detecting implied scenarios. In *4th International Workshop on Scenarios and State Machines: Models, Algorithms and Tool, ICSE 2005, St. Louis, Missouri, USA*, 2005.

[38] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Software*, 34(1):53–60, 2001.

[39] J. Grabowski. *Test Case Generation and Test Case Specication with Message Sequence Charts*. PhD thesis, Institute for Informatics and Applied Mathematics, Universitat Bern, 1994.

[40] T. Quatrani. *Visual modeling with Rational Rose 2000 and UML*. Addison Wesley, Reading, Mass., 1998.

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

[42] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: Current practice. *IEEE Software*, 15(2):34–45, 1998.

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

[44] C. Potts, K. Takahashi, and A.I. Anton. Inquiry-based requirements analysis. *IEEE Software*, 11(2):21–32, 1994.

[45] M.R. Young and P.B. Barnard. The use of scenarios in human-computer interaction research: Turbocharging the tortoise of cumulative science. In *Conference on Human Factors in Computing Systems and Graphics Interface (CHI/GI'87)*, pages 118–135, 1987.

[46] C. Rolland, C.B. Achour, C. Cauvet, J. Ralyte, A. Sutcliffe, N.A.M. Maiden, P. Haumer, K. Pohl, E. Dubois, and P. Heymans. *A Proposal for a Scenario Classification Framework*. CREWS Report 96-01, Cooperative Requirements Engineering

with Scenarios (ESPRIT Long Term Research Project 21.903), 1996. Available at http://Sunsite.Informatik.RWTH- Aachen.DE/CREWS/, 1996.

[47] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal approach to scenario analysis. *IEEE Software*, 11(2):33–41, 1994.

[48] P. Heymans and E. Dubois. Scenario-based techniques for supporting the elaboration and the validation of formal requirements. *Requirements Engineering Journal*, 3(3-4):202–218, 1998.

[49] G. Adriaens and D. Schreurs. From cogram to alcogram: Towards a controlled english grammar checker. In *14th International Conference on Computational Linguistics (COLING'92)*, pages 595–601, 1992.

[50] D. Jurafsky and J.H. Martin. *Context-Free Grammars for English (Chapter 9) in Speech and Language Processing*. Prentice Hall, Upper Saddle River, 2000.

[51] *ISO: Information processing systems - Open Systems Interconnection - Service conventions*. International Standards Organization, 1987.

[52] *CCITT: Interworking between the digital subscriber system layer 3 protocol and the signalling system 7 ISDN. User Part*. Recommendation Q.699, International Telephone and Telegraph Consultative Committee (CCITT), Geneva, 1988.

[53] *CCITT: Stage 2 of the method for characterization of services supported by an ISDN*. Recommendation Q.65, International Telephone and Telegraph Consultative Committee (CCITT), Geneva, 1988.

[54] *ITU: Message Sequence Charts*. Recommendation, International Telecommunications Union. Telecommunication Standardization Sector, 1992.

[55] *ITU: Message Sequence Charts*. Recommendation Z.120, International Telecommunications Union. Telecommunication Standardization Sector, 1993.

[56] *ITU: Message Sequence Charts*. Recommendation Z.120, International Telecommunications Union. Telecommunication Standardization Sector, 2000.

[57] J. Grabowski, P. Graubmann, and E. Rudolph. The standardization of message sequence charts. In *Proceedings of the Software Engineering Standards Symposium*, pages 48–63, 1993.

[58] I.H. Kruger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Institut fr Informatik, Technischen Universitat Mnchen, 2000.

[59] C. Szyperski. *Component Software. Beyond Object Oriented Programming*. Addison Wesley, Harlow, England, 1998.

[60] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, Harlow, 1999.

[61] R. Alur, G.J. Holzmann, and D. Peled. An analyser for Message Sequence Charts. In *2nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 35–48. LNCS 1055, 1996.

[62] A. Muscholl and D. Peled. Message Sequence Graphs and decision problems on Mazurkiewicz traces. In *24th International Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, pages 81–91. LNCS 1672.

[63] H. Ben-Abdhallah and S. Leue. *Expressing and Analyzing Timing Constraints in Message Sequence Charts*. Technical Report 97-04, Electrical and Computer Engineering, University of Waterloo, Waterloo, 1997.

[64] A. Engels, L. Feijs, and S. Mauw. MSC and data: dynamic variables. In *9th SDL Forum*, pages 105–120, 1999.

[65] O. Haugen. From MSC-2000 to UML 2.0 - the future of sequence diagrams. In *10th International SDL Forum*, pages 38–51. LNCS 2078, 2001.

[66] I. Kruger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *Distributed and Parallel Embedded Systems*, pages 61–71, 1999.

[67] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts '96. In *IFIP International Conference on Formal Description Techniques (FORTE'96)*, pages 1629–1641, 1996.

[68] D. Harel and W. Damm. LSCs: Breathing life into Message Sequence Charts. In *3rd IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 293–312, 1999.

[69] R. Alur and M. Yannakakis. Model checking of Message Sequence Charts. In *10th International Conference on Concurrency Theory (CONCUR'99)*, pages 114–129. LNCS 1664, 1999.

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

[71] S. Som, R. Dssouli, and J. Vaucher. From scenarios to timed automata: Building specifications from user requirements. In *Asia Pacific Software Engineering Conference (APSEC'95)*, pages 48–57, 1995.

[72] I. Khriss, M. Elkoutbi, and R.K. Keller. Automating the Synthesis of UML StateChart Diagrams from Multiple Collaboration Diagrams. In *UML'98: Beyond the Notation*, pages 132–147. LNCS 2078, 1999.

[73] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A compositional real-time semantics of STATEMATE designs. In *International Symposium on Compositionality: The Signifcant Difference* , pages 186–238. LNCS 1536, 1997.

[74] F. Maraninchi and N. Halbwachs. Compositional semantics of non-deterministic synchronous languages. In *6th European Symposium on Programming (ESOP'96)*, pages 235–249. LNCS 1058, 1996.

[75] H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki. SDE: Incremental Specification and Development of Communications. *IEEE Transactions on Computers*, 40(4):553–561, 1987.

[76] F. Bordeleau. *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*. PhD thesis, SCS, Carleton University, 1999.

[77] J. Klose and H. Wittke. An Automata Based Interpretation of Live Sequence Charts . In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 512–527. LNCS 2031, 2001.

[78] P.B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[79] H. Ben-Abdhallah and S. Leue. MESA: Support for scenario-based design of concurrent systems. In *4th Int'l Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, pages 118–135. LNCS 1384, 1998.

[80] S. Uchitel. *Incremental Elaboration of Scenario-Based Specifications and Behaviour Models Using Implied Scenarios*. PhD thesis, Imperial College, Londonn, 2003.

[81] K. Koskimies and E. Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software Practice and Experience*, 24(7):663–658, 1994.

[82] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *Distributed and Parallel Embedded Systems*, pages 61–71, 1998.

[83] D. Angluin. Learning regular sets from queries and counterexamples. *Journal of Information and Computing*, 75(2):87–106, 1987.

[84] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *International Journal of Foundations of Computer Science*, 13(1):5–51, 2002.

[85] I. Khriss, M. Elkoutbi, and R.K. Keller. Automatic synthesis of behavioral objects specifications from scenarios. *Integrated Design and Process Science*, 3(5):53–77, 2001.

[86] A.J. Mooij, N. Goga, and J.M.T. Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. In *Int'l Conf. on Fundamental Approaches to Software Engineering (FASE 2005)*, pages 273–288, 2005.

[87] A.J. Mooij, J.M.T Romijn, and W. Wesselink. Realizability criteria for compositional MSCs. In *11th International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 248–262, 2006.

[88] C. Fournet, C.A.R. Hoare, S.K. Rajamani, and J. Rehof. Stuck-free conformance. In *Computer-Aided Verification (CAV 04)*, pages 242–254, 2004.

[89] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[90] J. Thompson and M. Heimdhal. An integrated development environment for prototyping safety critical systems. In *IEEE Int. Workshop on Rapid System Prototyping*, pages 172–177, 1999.

[91] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Esterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *15th IEEE International Requirements Engineering Conference (RE'07)*, pages 221–230, 2007.

[92] M. Sabetzadeh, S. Nejati, S. Easterbrook, and M. Chechik. Global consistency checking of distributed models with TReMer+. In *30th International Conference on Software Engineering (ICSE'08)*, pages 815–818, 2008.