Synthesis of partial behavior models from overlapping scenarios with alternative alphabets

Mohammed Lafi

The University of Toledo

Follow this and additional works at: http://utdr.utoledo.edu/theses-dissertations

Recommended Citation
http://utdr.utoledo.edu/theses-dissertations/618

This Dissertation is brought to you for free and open access by The University of Toledo Digital Repository. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of The University of Toledo Digital Repository. For more information, please see the repository’s About page.
A Dissertation
entitled

Synthesis of Partial Behavior Models
from Overlapping Scenarios with Alternative Alphabets

by
Mohammed Lafi

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the
Doctor of Philosophy Degree in Engineering

Dr. Jackson Carvalho, Committee Chair

Dr. Mansoor Alam, Committee Member

Dr. Gerald Heuring, Committee Member

Dr. Henry Ledgard, Committee Member

Dr. Ivie Stein Jr., Committee Member

Dr. Patricia R. Komuniecki, Dean
College of Graduate Studies

The University of Toledo
December 2011
An Abstract of

Synthesis of Partial Behavior Models from Overlapping Scenarios with Alternative Alphabets

by

Mohammed Lafi

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the Doctor of Philosophy Degree in Engineering

The University of Toledo

December 2011

System requirements can be expressed as a set of scenarios and a set of properties. Behavioral models are synthesized from these scenarios and properties. Behavioral models can be used to validate requirements specification and during early stages of system design. Two problems arise during the synthesis process. They are overlapping scenarios, and alternative alphabets.

Scenarios can overlap and have shared actions. Synthesizing behavioral models from overlapping scenarios can result into redundant states and transitions. This research proposes the decomposition of the overlapping scenarios into a set of non-overlapping scenarios as a solution. It also proposes a technique to compensate overlapping scenarios by both identifying and decomposing them. It uses pattern matching to identify overlapping scenarios. Then, it introduces algorithms to decompose the overlapping scenarios. This research proposes the decomposition of the overlapping scenarios into a set of non-overlapping scenarios as a solution.

An alphabet is a set of actions that is used in scenarios and properties that describe a system. If the alphabet used before the application of synthesis algorithm is different from the alphabet after adding new requirements (scenarios or properties) then we call this situation alternative alphabets. This research introduces an approach that includes new alphabets in the generated behavioral models. Incorporating a new
alphabet to an existing behavioral model of property was achieved by adding a new term to the system alphabet. Then developing an algorithm that uses this term as a pointer to add the new alphabet discovered. Incorporating new alphabet to an existing behavioral model of scenarios was achieved by developing an algorithms that adds the new alphabet to transitions that go to a special state in the behavioral model, the sink state.
To my parents, Fayeq and Mazzoza, my wife, Abeer, and my kids, Ali, Noor, Salma, and Reema.
Acknowledgments

I could not have complete this work without the necessary assistance, guidance and support of my advisors, my previous advisor Dr. Hilda Standle y, and my current advisors Dr. Jackson Carvalho, and Dr. Mansoor Alam. Their comments and suggestions deeply influenced this work. My sincere gratitude goes to all of them.

Also, I would like to thank other members in my advisory committee Dr. Gerald Heuring, Dr. Henry Ledgard, and Dr. Ivie Stein Jr. for the time they devoted, out of their busy schedule, to read this dissertation and give helpful comments.

This dissertation would not have been possible without a generous scholarship from Alzaytoonah University of Jordan. I would like to thank Dr. Naser Saleh, Dr. Rushdi Ali Hasan, Dr. Abdel-Fatah Al-Tamimi, Dr. Mohammed Mohairat, and Mr Theyazan Qaqaa, for their continuous help and support.

I would like to thank Dr. Mohsin Jamali, Dr. Henry Ledgard, Dr. Charles J Odenthal, Dr. Ivie Stein Jr, and Dr. Thomas Stuart, for inspiring me in their excellent courses I have taken with them.

I would like to thank my parents Fayeq and Mazzoza for their love and support. Last but not least, I would like to thank my wife Abeer, and my kids Ali, Noor, Salma, and Reema for their love, patience, and sacrifices they have been made during the time that I prepared this work.
Contents

Abstract iii

Acknowledgments vi

Contents vii

List of Tables xii

List of Figures xiii

List of Abbreviations xvii

Preface xviii

1 Introduction 1
  1.1 The Problem .......................................................... 8
  1.2 Motivation ............................................................. 10
  1.3 The Solution ........................................................... 10
  1.4 Approach taken ....................................................... 11
  1.5 Thesis Overview ..................................................... 12

2 Basic Notions and Notations 14
  2.1 Scenarios .............................................................. 14
  2.2 Properties ............................................................. 16
    2.2.1 Fluents ............................................................ 17
7.3 Modifying the Partial Behavioral Model of the Properties to Reflect Newly Discovered Events

7.3.1 The Definition of The Alphabet Used in Properties

7.3.2 Comparing Two Types of Temporal Logic

7.3.2.1 Types of Temporal Logic

7.3.2.2 Closed (invariant) Under Stuttering Definition

7.3.2.3 What Types of Temporal Logic are More Appropriate to Use to Describe Properties (The Effect of Using the Next Operator)

7.3.3 The Proposed Algorithms

7.3.4 Example

7.3.4.1 The Effect of Adding a New Alphabet on the Behavioral Model of the Properties Without Using the Proposed Algorithm

7.3.4.2 The Effect of Adding a New Alphabet on the Behavioral Model of the Properties Using the Proposed Algorithm

7.4 Modifying the Partial Behavioral Model of the Scenarios to Reflect Newly Discovered Events

7.4.1 Modifying the Partial Behavioral Model of the Scenarios to Reflect Newly Discovered Events in Properties

7.4.2 Modifying the Partial Behavioral Model of the Scenarios to Reflect Newly Discovered Events in Scenarios

7.5 Conclusion

8 Summary and Conclusions

8.1 Summary
# List of Tables

6.1 Set of scenarios before decomposition. ........................................ 49  
6.2 Set of scenarios after decomposition. ........................................ 50  
6.3 Scenario Sc1 before decomposition. ......................................... 53  
6.4 Scenario Sc2 before decomposition. ......................................... 54  
6.5 Scenario Sc3 before decomposition. ......................................... 54  
6.6 Score matrix of scenarios Sc1 and Sc2 using pattern matching. .......... 55  
6.7 Tracing the score matrix of scenarios Sc1 and Sc2 to extract the overlapping part. ......................................................... 56  
6.8 Overlapping events between two scenarios, Sc1 and Sc2, using pattern matching. ......................................................... 57  
6.9 Overlapping scenarios set for scenarios Sc1 and Sc2 using pattern matching. 57  
6.10 Overlapping scenarios set after finding other scenarios that share the same overlapping parts. ........................................ 59  
6.11 Scenario Sc1 after decomposition. ......................................... 63  
6.12 Scenario Sc2 after decomposition ........................................... 63  
6.13 Scenario Sc3 after decomposition ........................................... 64  
6.14 Scenario NewSc that is generated after decomposition .................. 64
# List of Figures

1-1  Textual representation of a scenario that shows the interactions between a user and an ATM machine. ........................................ 2
1-2  An MSC representation of the scenario shown in Figure 1-1. .......... 3
1-3  The scenario in (b) contains additional actions to avoid the negative scenario characteristic in (a) ........................................ 4
1-4  A scenario, represented as MSC, that shows the interactions between user, cache and server. ........................................ 5
1-5  OCL representation of the constraints for the scenario shown in Figure 1-4. 5
1-6  Fluent representation of the constraints for the scenario shown in Figure 1-4. 5
1-7  Spiral model for software requirements process. .......................... 8

2-1  A generic scenario represented in bMSC notation. .................... 14
2-2  The hierarchy of the data structures ................................. 16

3-1  Given a set of scenarios prepared by the end-user and a set of properties prepared by a software engineer, a behavioral model for the proposed system is generated by a synthesis algorithm. .................... 23
3-2  Set of scenarios describing how to connect to an enhanced messaging system. 25
3-3  The behavioral model of the set of scenarios shown in Figure 3-2 with redundant states and transitions. ............................... 26
3-4  The behavioral model of the set of scenarios shown in Figure 3-2 without redundant states and transitions ............................... 26
4-1 Four values function. .................................................. 33

5-1 Create behavioral model for each instance. ................................ 39

5-2 Set of scenarios that illustrate the behavior of an alarm in a clock execution. 41

5-3 The behavioral models of the instances in Figure 5-2 .................. 42

5-4 The composition of the instance behavioral models shown in Figure 5-2 42

5-5 Conversion of a behavioral model into Partial behavioral model. .... 43

5-6 The behavioral model of two properties. .................................. 44

5-7 Creation of a behavioral model for properties without considering alternative alphabets. .............................................. 45

5-8 Conversion of a behavioral model of properties, represented as an LTS model into a partial behavioral model, represented as an MTS model. ... 45

6-1 Two scenarios represented in the MSC notation. ......................... 47

6-2 The proposed algorithm consists of two stages. In the first stage, which consists of four steps, it detects the overlapping scenarios using pattern matching. In the second stage, which consists of two steps, it decomposes the overlapping scenarios, and creates a new scenario for the overlapping part. ..................................................... 51

6-3 Creates the score matrix T for two scenarios Sc1 and Sc2. ............ 53

6-4 Extracts the overlapping events between scenarios Sc1 and Sc2 by tracing their score matrix. .................................................. 56

6-5 Finding the maximum overlapping part from the set of the overlapping parts. ................................................................. 58

6-6 Find other scenarios that have the same overlapping part. ............ 59

6-7 The proposed algorithm for the identification and decomposition of overlapping scenarios. ................................................. 61
6-8 The flowchart of the proposed algorithm that decomposes a set of overlapping scenarios. ....................................................... 62

6-9 Creates a new scenario by specifying the scenarios $Sc$ which has the overlapping part, the starting of the overlapping index and count the number of events in the overlapping part. ........................................ 62

6-10 In decomposing scenarios and adding join points, we have three cases: the overlapping part at the beginning of the scenario, the overlapping part at the end of the scenario, and the overlapping part in the middle of the scenario. Figures (a), (b), and (c) illustrate the different cases respectively. 66

6-11 Decomposing a scenario by creating a new scenario for the overlapping part and removing the overlapping part. The decomposition depends on the position of the overlapping part specified by index. It also depends on count which is the number of overlapping events. ....................... 67

7-1 A scenario, represented as an MSC, that shows the interactions between user, cache and server. ....................................................... 69

7-2 Constraints represented as an OCL on the scenario shown in Figure 7-1. 69

7-3 Constraints represented as fluents on the scenario shown in Figure 7-1. 70

7-4 Detailed synthesis algorithm diagram. ....................................................... 71

7-5 Modify the partial behavioral model of properties to incorporate the new actions that are discovered. ....................................................... 79

7-6 The modified synthesis algorithm. This algorithm generates a partial behavioral model, for both scenarios and properties if the new actions are discovered in properties. ....................................................... 80

7-7 Properties that describes a television system which blanks its screen while it tunes to a new channel. ....................................................... 80
7-8 The behavioral model of the set of properties without using the proposed algorithm. ................................................................. 81

7-9 Using the proposed algorithm, Figure (a) shows the behavioral model of the set of properties before the inclusion of the new alphabet, and Figure (b) shows the behavioral model of the set of properties after the inclusion of the new alphabet. ......................................................... 81

7-10 An algorithm that modify the partial behavioral model of scenarios to incorporate the new actions are discovered in properties. ............... 82

7-11 A partial behavioral model for a set of scenarios before addition of new alphabet, and a partial behavioral model for a set of scenarios after addition of new alphabet “e”, Figures (a), and (b) illustrate the first and second model respectively. ................................................................. 83

7-12 The modified synthesis algorithm. This algorithm generates a partial behavioral model, for both scenarios and properties if the new actions are discovered in scenarios. ......................................................... 83
List of Abbreviations

bMSC .................. basic Message Sequence Chart

DFA .................. Deterministic Finite Automata

FLTL .................. Fluent Linear Temporal Logic
FSM .................. Finite State Machine
FSP .................. Finite State Process

hMSC .................. high Message Sequence Chart

LCS .................. longest common sequence
LTS .................. Label Transition System

MSC .................. Message Sequence Chart
MTS .................. Modal Transition System

NFA .................. Non-Deterministic Finite Automata

OCL .................. Object Constraint Language

SD .................. Sequence Diagram
Preface

Developing a software application has many challenges. In the early stages of the software engineering discipline foundation, scholars and practitioners noticed that the development of a software application almost always exceeded both the budget and the time that was expected [1]. Also, they noticed that in many cases the final product did not meet the users’ expectations. In an attempt to reduce such problems, software engineers introduced the notion of engineering the process of software applications development [1]. As the result of such a modification the process of software development was structured as a set of stages. These stages include the requirements specification, the application design, the application implementations, the application test, and the application maintenance.

Requirements specification or requirements engineering is an important stage in the software development process. Any error happened during this stage can be costly, as it affects the other stages. Modifications during this stage are also costly because of the need to modify all products that result from stages that follow. Therefore, this stage needs to be designed carefully.

In the agile software development methods [2], the requirements specification, system design, system implementation, and testing are done during a repeated pattern, in short time periods. It is hard for the software engineers, the software developers, and users to maintain a traditional requirements document, in such a software process. An alternative approach to express the requirements specification is through the use of scenarios and properties. Scenarios and properties provide an intuitive way to
specify the system requirements [3, 4]. Synthesis algorithms that create a behavioral model representation for the scenarios and properties can be applied. The generated behavioral models can therefore be used to validate the requirements specification. Also, such representation may be used as an early design of the system.

This research focuses on the following two issues that occur during the behavioral model synthesis process, overlapping scenarios and alternative alphabets. Overlapping scenarios means the scenarios have common or shared parts. Overlapping scenarios can cause redundant states and transitions. This work introduces a technique to identify overlapping scenarios and decompose them.

Alternative alphabets occur when a new requirement (scenario or property) that has a new action or event is included. The approach proposed here modifies the current behavioral models to reflect the newly discovered alphabet instead of recreating the behavioral models.

After having represented the requirements specifications as scenarios and properties and having validated them through the generated behavioral models, it is expected that the resulting requirements specification will be closer to user’s expectations. We believe that relevant outcomes obtained by the use of the approach this work proposes are:

1. A decrease in the need for modification of the requirements specification,

2. A decrease in both the overall cost and the time required for the development of the application, and

3. An increase in the satisfaction of the user’s expectations.
Chapter 1

Introduction

A valid software development process produces useful software artifacts while controlling the cost and time needed for software production. The software development process consists of a set of phases depending on the type of software development model used (e.g. waterfall, incremental development, prototyping, spiral and agile) [5–7]. Typically, this process consists of the following phases [7]:

• definition -which produces user requirements specifications;

• concept -which produces design specifications;

• development- which produces beta software;

• evaluation- which produces release software; and

• operation- which produces operational software.

The definition phase is the subject of this research. This phase is also referred to as requirements engineering [5].

Requirements engineering is applied through a thorough analysis of the services to be provided by the proposed software system. This process typically exposes operational constraints that are not to be violated in the system design. The requirements engineering process can easily generate a great amount of detailed information that
can be confusing and difficult to understand. As illustration the requirements document, which is the document that contains the user requirements specifications, may easily contain hundreds of pages [5]. A more effective way to express the user requirements specification can be done by writing the user requirements as *scenarios*, eliminating much of the excess details that is found in the requirements document.

*Scenarios* are written descriptions used to organize the information generated by the requirements analysis [8]. These descriptions are typically in *story* form giving a sequence of events describing the system’s behavior. Representing user requirements as a set of scenarios is an established activity in software engineering [3, 4] and its use makes the communication between end users and software developers more efficient [9]. This is because the representation used facilitates its understanding by all stakeholders with different technical levels [9]. Figure 1-1 [10] illustrates the use of a scenario to describe an interaction between an Automated Teller Machine (ATM) in a situation when a user enters an incorrect password.

```
The ATM asks the user to insert a card.
The user inserts a cash card.
The ATM accepts the card and reads its serial number.
The ATM requests the password.
The user enters 1919.
The ATM verifies the serial number and password with the consortium.
The consortium rejects the password after consulting the appropriate bank.
The ATM indicates a bad password and asks user to re-enter it.
The user hits cancel.
The ATM displays cancel message.
The ATM prints a receipt, ejects the card, and asks the user to remove both of them.
The user takes the receipt and the card.
The ATM asks a user to insert a card.
```

Figure 1-1: Textual representation of a scenario that shows the interactions between a user and an ATM machine.

To assist with representing scenarios in intuitive and clear forms, *sequence diagrams*,
such as the Unified Modeling Diagrams (UML) [8] and Message Sequence Chart (MSC) [11], are used. For instance, Figure 1-2 [10] shows the basic Message Sequence Charts (bMSC) corresponding to the ATM example of Figure 1-1.

A negative scenario is a scenario that must be avoided in the system under design. The purpose of negative scenarios is to identify potentially harmful actions taken by the software product [12]. Scenarios are appropriate for eliciting functional requirements. On the other hand, negative scenarios are appropriate for eliciting non-functional requirements. In addition, negative scenarios give justification to design decisions and subsystems that may be added to handle exceptions [12].

For example, Figure 1-3(a) shows a negative scenario. In this scenario, the ATM tries to verify an account information with consortium, but there is no connection between ATM and consortium. Figure 1-3(b) shows a modification of the negative scenario to consider testing the connection before trying to verify the password. The addition of the new exception case is justified by the negative scenario. Figure 1-4 [13] shows another negative scenario. This scenario shows the interactions between the
user, the cache and the server. When a user requests data from the cache, the cache requests data from the server and then the cache responds with the data to the user. If the user requests data from the cache again, the cache responds immediately without requesting data from server. But if the data is changed in the server, the cache should not respond with its outdated copy. So, the last action ResponseCacheData must be avoided.

In addition to scenarios, properties can be used to add constraints and/or conditions that limit the behavior of a system [14, 15]. Properties are used to ensure that the system will not exhibit incorrect or disallowed behavior. Properties are often represented by a formalism such as the one proposed by the Object Constraint Language (OCL) [16] and Fluent Linear Temporal Logic (FLTL) [15]. For instance, Figure 1-5 [13] shows the OCL representation for the example in Figure 1-4 [13]. As it can be seen it contains the values of system variables before and after the executions of the messages. The FLTL representation of the properties for the same example is shown in Figure 1-6.

To validate the software requirements, expressed as a set of scenarios and a set of properties, a behavioral model is developed [9]. A behavioral model is a formalism
Figure 1-4: A scenario, represented as MSC, that shows the interactions between user, cache and server.

```plaintext
requestCacheData
pre: requestPending = false
post: requestPending = true

responseCacheData
pre: requestPending = true
and cached = true
post: requestPending = false

requestServerData
pre: requestPending = true
and cached = false
post:

responseServerData
pre: post: cached = true
```

Figure 1-5: OCL representation of the constraints for the scenario shown in Figure 1-4.

```plaintext
Fluent requestPending=<requestCacheData, responseCacheData> initially false
Fluent cached=<responseServeData, WriteData> initially false
```

Figure 1-6: Fluent representation of the constraints for the scenario shown in Figure 1-4.
that describes the behavior of the system using a sequence of state transitions [9]. It provides a graphical representation for the new system to be developed [9]. For human understanding of a graphical representation is clearer than for textual presentation in the description of the operation of the system. Therefore, it is easier to understand and validate by both users and software engineers [5, 9].

A behavioral model uses a set of states and a set of state transitions. The states represent all possible situations where the system can be. The transitions determine how the system moves from one situation to another. In the behavioral model, two types of transitions are modeled, allowed transitions and disallowed transitions. Allowed transitions are shown explicitly in the model, and disallowed transitions are expressed implicitly in the model by the absence of a related transition. At the start of developing a system model, many decisions are not yet made regarding allowed and disallowed transitions. The may-be transition is used to model the transitions of uncertain status. The may-be transition is characterized by the addition of the symbol ? at the end of the transition label. A behavioral model, which includes may-be transitions, is called a partial behavioral model [14].

In order to convert a set of scenarios to a behavioral model, synthesis algorithms can be used. A number of synthesis algorithms are already available [14, 17–20]. However, such algorithms share two characteristics that are not well addressed or not addressed at all. They are overlapping scenarios and alternative alphabets.

Overlapping scenarios are scenarios that have shared parts. One limitation of such scenarios is the fact that they require the synthesis algorithm to create a behavioral model with redundant states [21]. This research proposes the decomposition of the overlapping scenarios into a set of non-overlapping scenarios as a solution.

An alphabet is a set of actions that is used in scenarios and properties that describe a system. Alternative alphabets, for a set of scenarios or a set of properties, arise when new actions, in the new scenario or property, are discovered and need to be
included. In other words, consider, for instance, \( \alpha_1 \) to be the alphabet that was used during the creation of a partial behavioral model and \( \alpha_2 \) to be the alphabet \( \alpha_1 \) union the new actions discovered when a new scenario or property is included. We call \( \alpha_1 \) and \( \alpha_2 \) alternative alphabets.

Modifying the partial behavioral model, when alternative alphabets are considered, presents a problem\(^1\), and in this case, there are two possible options for solutions. We could either

(1) restart the synthesis algorithm from the beginning, and create a new model or

(2) modify the existing model [14].

Alternative (2) avoids the exponential growth that results from the computation time of synthesizing a partial behavioral model from a larger number of scenarios and properties when an alphabet with a large number of elements is considered [22].

In the following, we explain the importance of using scenarios and properties in the software requirements process and their relationship with the requirements document.

The requirements engineering process consists of a set of activities. These activities are requirements elicitation, requirements analysis, requirements definition, requirements prototyping, requirements reviews, and requirements specification and documentation [23]. Figure 1-7 shows a spiral model of these activities [24]. Requirements can be represented as scenarios and properties in all of the activities during the requirements engineering process activities [23]. Also, representing requirements as scenarios can be considered as prototyping. In addition, scenarios and properties can be a main component in the software specification document [23]. Although scenarios and properties are known to be used in describing requirements, they can also be used both during the early stages of the design and in preparing test cases for the developed application [25].

\(^1\)The problem is that the synthesis algorithm assumes that the alphabet is fixed. However, in the case of alternative alphabet it is not.
The common way for describing requirements specification is using the disciplined document in structured natural language. However, the diagrammatic notation, such as MSC, is currently in use for this purpose [23]. Such forms of notations are used as a complement and not a replacement for requirement document. The reason for this is because they require descriptions in natural language to give more details and explanation of the requirements. In the next sections, we outline this work and the problem addressed by the proposed solution.

1.1 The Problem

This research address the synthesis of behavioral models in the requirements engineering stage of software development. It particularly focuses on two problems: overlapping scenarios and alternative alphabets.

(1) Overlapping Scenarios

Scenarios are usually provided by different stakeholders. Stakeholders usually have different roles and prospectives. Therefore scenarios usually have overlapping parts. This means two or more scenarios have shared parts. The creation of the behavioral model for such scenarios will include additional states and transitions. For this reason the resulting model is considered sub-optimal.

(2) Alternative Alphabets.
Alphabet is a set of actions exhibited by scenarios and properties. New scenarios and properties usually appear during the requirements engineering process. These scenarios and properties usually have new alphabets that were not considered before the generation of the behavioral model. We call this situation an alternative alphabet, where the alphabet before the creation of the behavioral model is different than the alphabet after the creation of the behavioral model. In other words, alternative alphabets are caused by the difference between the alphabets, before and after the generation of the behavioral model, as a result of adding a new property or a new scenario to the requirements set. To include the new alphabet in the behavioral models, synthesis algorithms are required to restart.

In order to approach both of the above, we consider the following issues:

- There are no errors in the representation of actions
- Scenarios, properties, behavioral models, partial behavioral models are represented using MSC, FLTL, LTS, and MTS notations, respectively.
- The conversion algorithms are the algorithms described in [21] and in [14].
- Merging models will not be considered in this thesis.
- Any new event is either originated from a scenario or a property. When a new event is added, the designer may be faced with the following semantics.

  1. Modifying the partial behavioral model properties to reflect newly discovered events in properties.
  2. Modifying the partial behavioral model of scenarios to reflect newly discovered events in properties.
3. Modifying the partial behavioral model of properties to reflect newly discovered events in scenarios.

4. Modifying the partial behavioral model of scenarios to reflect newly discovered events in scenarios.

This research considers cases 1, 2, and 3. Case 4 will not be considered in this research.

1.2 Motivation

This work was motivated by the wide use of scenarios and properties to describe user requirements [3,4] and the need of synthesis behavioral models from them [14,17–20] to validate these user requirements. As noted in [21] scenarios can overlap, having shared parts. Also as noted in [14] new requirements, represented as scenarios and properties, can be added after the creation of the behavioral models that contain a new alphabet. The necessity of identifying shared parts in scenarios and having behavioral models without extra states and transitions motivated the research reported in this work. The need of the ability for modifying behavioral models without reconstructing them when new alphabet is discovered also motivated this work. The section that follows outlines two solutions, one for the overlapping scenarios based on pattern matching and decomposing scenarios, and the other for alternative alphabets based on updating behavioral models instead of recreating them.

1.3 The Solution

We propose that overlapping scenarios can be detected using pattern matching and then can be decomposed into a set of scenarios such that overlaps are reduced. We also propose that behavioral models can be modified, when alternative alphabets
occur. In this thesis, we introduce a pattern matching algorithm to detect overlapping scenarios and algorithms to decompose the overlapping scenarios into a set of scenarios with reduced overlaps. Also, we introduce algorithms to modify behavioral models, instead of recreating them, when alternative alphabets occur. We claim the following:

- Overlapping scenarios can be detected using pattern matching,
- Decomposing scenarios can reduce the overlaps between scenarios,
- Detection and decomposition of the overlapping scenarios lead to more streamlined behavioral models, and
- Modifying behavioral models instead of recreating them can be achieved in the case of alternative alphabets.

### 1.4 Approach taken

This section outlines the approach taken to address both the overlapping scenarios and alternative alphabets problems.

- **Overlapping Scenarios:**
  A pattern matching-based method to identify and decompose overlapping scenarios is proposed. The approach uses dynamic programming to extract the overlapping parts between scenarios. It also introduces a set of algorithms to support both the identification and the decomposition of the overlapping scenarios.

- **Alternative Alphabets:**

  In the behavioral model of properties, there are two sources of transitions. The
first is the communicating alphabet of the property, which is the source of allowed transitions. The second is the difference between the superset alphabet and the communicating alphabet of the property, which is the source of the *may be* transitions. Any new alphabet added to the existing alphabet can affect only the transitions that were originated from the second source. Therefore, we develop an algorithm to use this characteristics to include the new alphabets in the generated behavioral model of properties.

In the behavioral model of scenarios, there are two sources of the *may be* transitions. The first is the difference between communicating alphabet and the outgoing transitions from a state. The second is the difference between the superset alphabet and the communicating alphabet. The *may be* transitions go from each state to a special state called *sink* state. We use the above characteristic to develop an algorithm to include new alphabets in the generated behavioral model of scenarios.

## 1.5 Thesis Overview

In this thesis, we start by introducing both overlapping scenarios and alternative alphabets. Then, we provide the necessary notation and explain the previous related work with special emphasis on algorithms that we relate our work to. After that, we propose solution for two issues that we are considering in this thesis: overlapping scenarios and alternative alphabets. Finally, we give summary and conclusions.

This thesis is organized as follows. Chapter 3 introduces the problem statement. Chapter 4 reviews the research literature that relates to synthesizing behavioral models from a set of scenarios and a set of properties. It focuses on how existing approaches deal with overlapping scenarios and alternative alphabets. Chapter 2 introduces the necessary background on scenarios notations, properties notations, and
behavioral models notations. Knowledge of this background is important to understand the rest of this dissertation. Chapter 5 explains the algorithms used to synthesize both the behavioral model and the partial behavioral model from scenarios and properties. Chapter 6 describes a proposed solution for the overlapping scenarios problem. It also discusses pattern matching and shows how the overlapping scenarios can be detected using this approach. Finally, it describes how to decompose overlapping scenarios and how to create a new scenario for the overlapping part. Chapter 7 defines the notation of alternative alphabets and illustrates the modification of behavioral models to reflect newly discovered events. Also, it shows how to modify the partial behavioral model of the properties, and partial behavioral model of scenarios to reflect newly discovered events. Summary and conclusions are given in Chapter 8.
Chapter 2

Basic Notions and Notations

This chapter introduces the basic notations that are used through this work. Section 2.1, Section 2.2, and Section 2.3, introduce scenarios notations, properties notations, and behavioral models notations, respectively.

2.1 Scenarios

Figure 2-1: A generic scenario represented in bMSC notation.

Figure 2-1 shows a generic scenario represented in the bMSC notation. In this figure:

- P, Q, and R, the vertical arrows, are instances. Instances represent users and system components. The length of the vertical arrows indicates the passage of time [21].
• M and N, the horizontal arrows, are actions/messages. Actions/messages represent interactions between instances. The direction of the arrow represents the direction of the interaction between the sender (tail of arrow) and the receiver (head of the arrow) [21].

The following sets are used to describe the algorithms in this dissertation.

• Instance is a component or actor in a scenario. We will use instances to represent the set of components or actors in the scenarios set.

• Actions represent the interactions between instances. We will use actions to represent the set of actions in both the scenarios and the properties set.

• Each event consists of three fields: sender, action, and receiver. Sender and receiver are instances. We will use events to be the set E such that E ⊆ (Instances × Actions × Instances).

• We will use scenario to represent a set of ordered events that has a name.

• A join point is a point that connects two scenarios.

• We will use scenariosSet to represent a set of Scenarios.

Figure 2-2 shows the hierarchy of the sets described above. The following are examples of these sets taken from the general scenario shown in Figure 2-1 where Instances = {P, Q, R}, Actions = {M, N}, Events = {(P, M, Q), (Q, N, R)}, Scenarios = {Sc1, {1:(P, M, Q), 2:(Q, N, R)}}, ScenariosSet = {Sc1}. In the algorithms described in this dissertation, we will use Sc[i] to refer to scenario i in a scenario set, Sc.events[i] to refer to i-th event in the scenario Sc, Sc.events[i].action to refer to the action in the i-th event in the scenario Sc, Sc.events[i].sender to refer to the sender in the i-th event in the scenario Sc, and Sc.events[i].receiver to refer to the receiver in the i-th event in the scenario Sc.
Definition 1 (Overlapping Scenarios). Let scenario \(sc\) be a sequence of events \(e_1, e_2, e_3, \ldots, e_n\), where \(n\) is the number of events in the scenario \(sc\). Let \(OV\) be a sequence of events \(v_1, v_2, v_3, \ldots, v_m\), where \(m\) is the number of events in the sequence \(OV\). We say that the sequence \(OV\) is an overlapping part in \(sc\) if there exists a monotonically increasing sequence of integers \(k_1, k_2, \ldots, k_m\) such that: \(e_{k_i} = v_i\), where \(1 \leq i \leq n\). We say that sequence \(OV\) is an overlapping part between \(sc_1\) and \(sc_2\), if the sequence \(OV\) is an overlapping part between both \(sc_1\) and \(sc_2\) [26].

2.2 Properties

This section explains the notation used to describe system properties. System properties can be either safety properties, which are properties that assert nothing bad will happen or liveness properties which are properties that assert good things will
happen [27, 28]. In this work, we consider the safety properties only\(^1\). It is necessary to understand how system properties can be represented in order to understand the synthesis algorithms.

In Section 2.2.1, we define the fluents. Then, in Subsection 2.2.2, the Fluent Linear Time Temporal Logic (FLTL) formalism is defined. Finally, in Subsection 2.2.3, we explain how to use the FLTL formalism to describe properties.

### 2.2.1 Fluents

The term *fluent* was proposed initially in the domain of artificial intelligence knowledge representation [30]. Later, it was adopted in the representation of properties in event-based systems [15, 31].

**Definition 2 (fluent).** A fluent is a proposition that has a varying value over time. The value of a fluent is true if it is initiated by an initial event at some earlier time and not terminated by a terminating event. The value of a fluent is false if it is terminated by a terminating event and not initiated after that [15].

The syntax used to represent fluent is \( \text{Fluent } \text{name} = < \text{ievents}, \text{tevents} > \) initially \( \text{i-value} \) where, \( \text{name} \) is the name of the fluent, \( \text{ievents} \) is the initiating event of the fluent, \( \text{tevents} \) is the terminating events, \( \text{ievents} \) is the initiating event of the fluent, and \( \text{i-value} \) is the initial value. We illustrate the concept of fluent by means of the two examples. Example 1, taken from [18], which shows the fluents describing a safety property “*keep the train’s doors closed while the train is moving.*” Example 2, taken from [15], which shows the fluents describing the “*blanking of the television screen*” and the fluent describing the “*tuning of a television channel.*”

\(^1\)The reason for this is because safety properties can be converted into a behavioral model and liveness property cannot [29].
Example 1 (Fluents). The fluents describing a safety property “keep the train’s doors closed while the train is moving” are:

Fluent moving = <{start}, {stop, emergency-stop}> initially false.
Fluent doors-open = <{open-doors, emergency-open}, {close-doors}> initially false.

Example 2 (Fluents). The fluent describing the blanking of the television screen is:
Fluent BLANKED = < blank, unblank >
and the fluent describing tuning a television channel is:
Fluent TUNING = < tune, endtune >

2.2.2 The Fluent Linear Temporal Logic (FLTL) Formalism

The FLTL formalism is used for reasoning about fluents [15]. Also, it is used to describe the properties of the system that add constraints on the system behavior that should follow during the running of the system over time.

The FLTL formalism is a fluent or an expression of fluents defined using the standard boolean operators and temporal operators. In this work we use the symbols ∧, ∨, and ¬ for the boolean AND, OR, and NOT operators, respectively. We also use the symbols X, U, W, F, ◇, and □ for the next, strong until, weak until, eventually (finally), and always (globally) operators, respectively [15].

Definition 3 (Satisfaction operator ). Let ϕ be an expression of the FLTL formula, and let S be a sequence of events. We say that S satisfy ϕ denoted by S |= ϕ, if all events in S satisfy the property ϕ [9, 15].

Definition 4 (FLTL formula). Let φ be a set of fluents and let p, q be fluents. Then an FLTL formula is p, ¬p, p ∧ q, p ∨ q, Xp, ◇p, □p, pUq, pWq.

Definition 5 (The meaning of an FLTL expression). The meaning of an FLTL expression can be defined as follows: Let T be an infinite trace\(^2\), and the satisfaction

\(^2\text{Trace, also known as a run, is a set of events to be performed [15].}\)
operator \((T, i) \models p\) means that the FLTL expression \(p\) is satisfied by trace \(T\) at event \(i\).

The meaning of the standard boolean operators and temporal operators are defined as follows [9]:

\[(T, i) \models Fl \iff (T, 0) \models Fl\]
\[(T, i) \models \neg q \iff \neg((T, i) \models p)\]
\[(T, i) \models p \land q \iff ((T, i) \models p) \land ((T, i) \models q)\]
\[(T, i) \models p \lor q \iff ((T, i) \models q) \lor ((T, i) \models q)\]
\[(T, i) \models Xq \iff (T, i + 1) \models q\]
\[(T, i) \models \Diamond p \iff \exists j \geq i \land (T, j) \models p\]
\[(T, i) \models \Box p \iff \forall j \geq i \land (T, j) \models p\]
\[(T, i) \models pUq \iff \exists j \geq i\) such that \((T, j) \models q \land \forall k, i \leq k \leq j(T, k) \models p\]
\[(T, i) \models pWq \iff (T, i) \models (pUq) \lor (T, i) \models \Box p\]

### 2.2.3 The Use of the FLTL Formalism to Describe the Properties of a System

The FLTL formalism can be used to describe the properties of a system. We illustrate this by the two examples taken from [32] and [15], as follows. Example 3 describes the system properties of a simple webmail application and Example 4 describes the property of a television system whose screen should be blank while switching to a new channel.

**Example 3 (Properties).** The system properties of a simple webmail application defined using the FLTL formalism.

Registered = < enable, disable > initially True
Logged\text{In} = < authenticate, \{ logout, disable \} > initially false

(legal access) \(P_1 = \Box (\text{loggedIn} \Rightarrow \text{Registered})\)

(private access) \(P_2 = \Box (\text{sendMsg} \Rightarrow \text{Logged\text{In}})\)
(logouts and ackd) \( P3 = \Box (\text{logout} \Rightarrow X\text{logoutMsg}) \)

**Example 4 (Property).** The property describing a television system which its screen should be blank while switching to a new channel.

\( NOARTIFACTS = \Box (\text{TUNING} \Rightarrow \text{BLANKED}) \)

### 2.3 Behavioral Model

In this section, we provide a set of definitions for Labeled Transition System (LTS) [21], Modal Transition System (MTS) [14], parallel composition [21], Buchi automata [33], and accepting trace [33]. We provide these definitions for the sake of completeness.

**Definition 6** (Labeled Transition System (LTS) [14]). Let \( U \) be a universal set of states, and \( \text{Act} \) be a universal set of observable actions\(^3\) labels. An LTS is a tuple \( L = (S, \text{act}, \Delta, s_0) \), where \( S \subseteq U \) is a finite set of states, \( \Delta \subseteq (S \times \text{Act} \times S) \) is a transition relation, and \( s_0 \in S \) is the initial state. We say that model \( L \) is transited to model \( L' \) denoted by \( L \xrightarrow{\text{action}} L' \), if \((s, \text{action}, s') \in \Delta \) and \( L' = (S, \text{act}, \Delta, s'_0) \).

**Definition 7** (Modal Transition System (MTS) [14]). An MTS \( M \) is a tuple \( M = (S, \text{act}, \Delta_1, \Delta_2, s_0) \), where \( \Delta_1 \subseteq \Delta_2 \), \((S, \text{act}, \Delta_1, s_0) \) is an LTS that represents the required\(^4\) transitions, \((S, \text{act}, \Delta_2, s_0) \) is an LTS that represents and the possible\(^5\) transitions.

**Definition 8** (Parallel composition [21]). Let \( M_1 \) and \( M_2 \) be two LTS models, and let \( M_1 = (Q_1, A_1, \Delta_1, q_{01}) \), and \( M_2 = (Q_2, A_2, \Delta_2, q_{02}) \). Then the parallel composition

---

\(^3\) Observable action is an action that can be seen and recognized by the externally entities or objects.

\(^4\) A required transition is a transition that must exit in the model.

\(^5\) A possible transition is a transition that can or cannot exist in the model in the model but, the decision about that is not taken yet.
operation $\parallel$ for $M_1$ and $M_2$ is defined as follows. $M_1\parallel M_2 = (Q, A, \Delta, q_0)$ where $Q = Q_1 \times Q_2$, $A = A_1 \cup A_2$, $q_0 = (q_{01}, q_{02})$, and $\Delta$ is defined as follows

$$
\frac{M_1 \overset{a}{\rightarrow} M_1, a \notin A_2}{M_1\parallel M_2 \overset{a}{\rightarrow} M'_1\parallel M_2} \quad (2.1)
$$

$$
\frac{M_1 \overset{a}{\rightarrow} M_1, M_2 \overset{a}{\rightarrow} M_2, a \notin \tau}{M_1\parallel M_2 \overset{a}{\rightarrow} M'_1\parallel M'_2} \quad (2.2)
$$

where $a$ as an observable action or $\tau^6$.

**Definition 9** (Buchi automata [33]). Let $S$ be a finite set of states, $E$ be a finite alphabet, $s_0 \in S$ the initial state, $T : S \times E \rightarrow S$ the transition function, and $F \subseteq S$ the final (accepting) states. We call the automata $B = (S, E, s_0, T, F)$ a Buchi automata.

**Definition 10** (Accepting trace [33]). Let seq be the infinite sequence $s_0, s_1, s_2, \ldots, s_n$. We say that seq is an accepted trace by Buchi automata $B$ if the following conditions are satisfied:

1. $s_0 \in S$ is the initial state,

2. $s_n \in F$ for infinitely many times (infinitely often), and

3. for each $s_i$, $i > 0$, $s_{i+1} \in T(s_i, e)$ and $e \in E$.

$^6\tau$ Non-observable actions are the actions which can only be seen and recognized by the entity or the object itself.
Chapter 3

Problem Statement

In the software engineering process, user requirements specifications can be represented as sets of scenarios and properties. Scenarios are written descriptions used to organize the information generated by requirements analysis, and properties are constraints or conditions that limit the behavior of a system.

In order to validate the user requirements specifications, a software engineer creates a behavioral model which is a graphical representation of the system that describes its behavior. This process is done by the application of the following steps. First, the end user states the system requirements as a set of scenarios that describes how the system works. Next, the software engineer adds a set of properties that ensures the system works correctly. Synthesis algorithms may then be applied to convert these requirements into a behavioral model that comprehensively describes the intended system behavior. This sequence of events is represented in Figure 3-1.

Existing synthesis algorithms [14,21] poorly address two commonly occurring characteristics of scenarios which are overlapping scenarios and alternative alphabets.

Unresolved overlapping scenarios (when scenarios have actions in common) can cause excess states when the model is processed by the synthesis algorithm. For example, if there are $n$ actions shared between $m$ scenarios then $(n + 1)(m - 1)$ redundant states would result. The reason for this is because if the same actions
Figure 3-1: Given a set of scenarios prepared by the end-user and a set of properties prepared by a software engineer, a behavioral model for the proposed system is generated by a synthesis algorithm.

occur in \( m \) scenarios, then there are \((m - 1)\) extra copies of these actions. Also, to represent \( n \) actions \((n + 1)\) states are required.

In the absence of a method to handle differing or alternative alphabets (whenever a new set of actions, described by a new alphabet, is added), existing synthesis algorithms must be rerun with the additional input. Since the computational time-complexity of the synthesis algorithms is exponential [22], even with contemporary computer processor speeds, a rerun could cause significant and costly delays to the project.

A synthesis algorithm would benefit from two extensions: (1) to address overlapped scenarios [21] and (2) alternative alphabets [14]. Section 3.1 and Section 3.2 explain these two issues, respectively.

### 3.1 Overlapping Scenarios

A set of scenarios is said to be non-overlapping if they do not share common actions. On the other hand, for scenarios to overlap there must be shared actions between two or more of them, where one or more actions appear in more than one scenario [21].

Overlapping scenarios can happen in patterns such as prefix-prefix, postfix-postfix, and postfix-prefix [34]. It can happen in other patterns such as prefix-infix, infix-infix,
and postfix-infix. This work introduces algorithms to detect all of the above listed overlapping patterns. Chapter 6 addressed this problem.

Scenarios are provided by different users with different viewpoints. It is expected that they are likely to overlap, therefore having shared parts. However, this overlapping results in a behavioral model with redundant states, because the synthesis algorithm will generate additional states for the overlapped parts [21]. Therefore, eliminating overlapped parts from scenarios can create a more stream-lined behavioral model.

Figure 3-2, for example, shows a set of scenarios which describes how to connect to a messaging center [35]. The user can call the messaging center by either dialing the messaging center number from his/her own phone, or from unsubscribed phone, or from another subscribed phone. Figure 3-3 shows the behavioral model of the set of scenarios in Figure 3-2. As it can be seen this behavioral model contains redundant states and transitions. Figure 3-4 shows the behavioral model of the set of scenarios in Figure 3-2 without the redundant states and transitions.

3.2 Alternative Alphabets

Each scenario exhibits a set of actions which may be described as a communicating alphabet. We will consider the set of actions recognized by all scenarios and call it the superset alphabet. Similar scenarios may be described by different alphabets originating from different stakeholders since individual viewpoints and vocabularies can be expected to differ [14].

Usually a synthesis algorithm is used to build a partial behavioral model from a set of scenarios and a set of properties incrementally. As a consequence, it is common that new scenarios and new properties are added during the requirements engineering and model building stages [14]. In many times, these new scenarios and
(a) Sc1: subscriber call to EMS from his phone.

(b) Sc3: subscriber call to EMS from another subscriber’s phone.

(c) Sc2: subscriber call to EMS from unsubscribed phone.

(d) Sc4: The caller hangs up.

(e) Order of execution for scenarios (a), (b), (c), and (d)

Figure 3-2: Set of scenarios describing how to connect to an enhanced messaging system.
Figure 3-3: The behavioral model of the set of scenarios shown in Figure 3-2 with redundant states and transitions.

Figure 3-4: The behavioral model of the set of scenarios shown in Figure 3-2 without redundant states and transitions.
new properties have new actions that were not considered in a previous application of the synthesis algorithm. Therefore, it is important to find a way to incorporate the newly discovered actions into the set of the known actions [14].

The existing synthesis algorithms do not take into consideration two aspects: overlapping scenarios and alternative alphabets [14, 21]. A synthesis algorithm may accommodate overlapping scenarios by eliminating the overlapped parts. Such a characteristic will reduce the required number of states. Also, a synthesis algorithm should accommodate alternative alphabets by reflecting the effect of new actions on the existing states of the behavioral model. This will eliminate resource expenditures in restarting the synthesis algorithm from the beginning.

3.3 The Context of the Work

In this section, we discuss the constraints and the scope of this research. During the development of a software application, the requirements can be represented as a set of scenarios and a set of properties. One way to validate these requirements is to build a behavioral model. However, it is difficult and time-consuming to build such a component. Algorithms have been proposed to automate the conversion from scenarios and properties to behavioral models. This research focuses on two issues in these conversion algorithms, scenarios overlapping and alternative alphabets.

When addressing scenario overlapping, we assumed that there are no errors in the representation of actions. We also assumed that during the development of these scenarios, a glossary for these actions will be available. This glossary prevents representing one action more than once. The action, say for example, *dial password* will not appear as *enter password*, providing an alternative representation to the same meaning. In this regard is to adjust common actions that occur in two or more scenarios. As a consequence, the approach introduced in this research will result in more
optimal behavioral model. The inputs of the synthesis algorithms are scenarios and
the outputs of it are behavioral models. The proposed approach improve scenarios by
removing overlaps. As a consequence, behavioral models are improved. Our focus is
go generate a more optimized behavioral model from scenarios with less overlaps and
not to generate a behavioral model from scenarios with overlaps then try to optimize
the generated model.

In addressing alternative alphabets the method proposed in this work considers
the addition of new actions to the alphabet after the creation of the behavioral model.

We consider the following assumptions, which are required by most synthesis
algorithms.

- Scenarios are presented as Message Sequence Charts (MSC) (see Section 2.1 on
page 14).

- Properties are represented as Fluent Linear Temporal Logic (FLTL) (see Sec-
tion 2.2 on page 16).

- The behavioral models are represented as Labeled Transition System (LTS) (see
Section 2.3 on page 20).

- Partial behavioral models are represented as Modal Transition System (MTS)
(see Section 2.3 on page 20).

- The conversion algorithms are the algorithms described by Uchitel(2003) et al.
in [21] and Uchitel(2009) et al in [14] (see Section 5.1 on page 38 and Section 5.2
on page 40).

Merging partial behavioral models will be out of the scope of this study.
Chapter 4

Background

This chapter gives the details of some of important algorithms that are used to synthesize behavioral models from scenarios and properties. We give remarks on the research literature that relates to synthesizing behavioral models from a set of scenarios and a set of properties. These remarks focus on how existing approaches treat overlapping scenarios and alternative alphabets. Although most of these approaches do not consider overlapping scenarios and alternative alphabets, some of them refer to these two aspects as possible extensions to their work.

4.1 Whittle and Schumann

The inputs to Whittle and Schumann [17] algorithm are sequence diagrams (SDs) and a set of constraints represented in OCL notation as pre- and post-conditions. The outputs of this approach are statecharts and UML class diagrams. The approach requires that the requirement specification should include the identification of global state variables, representing important aspects of the system. Also, it requires that the pre- and post-conditions refer to the global state variables. The values of the global state variables are stored in a list called a state vector. This approach consists of three steps. First, for each SD, the algorithm places the state vector before and after each message. The values of the state vector are determined by the message and both
pre- and post-conditions. Since, the initial state vector contains many unknowns, the algorithm propagates the values of the state vector using two methods: unification, and axioms. In the unification method two state vectors, \( v_1 \) and \( v_2 \), are considered the same if there exists a variable assignment \( \phi \), such that \( \phi(v_1) = \phi(v_2) \). In other words, there is some condition that enforces the values of \( v_1 \) and \( v_2 \) to be the same. In the axiom method, the state vector after a message and a state vector before the next message are assumed to have the same values. To illustrate this consider, for example, the sequence: (Message \( i \), State vector \( j \), State vector \( j + 1 \), Message \( i + 1 \)). Both the state vector \( j \) and the state vector \( j + 1 \) should have the same values. If not, then there is a conflict which should be resolved before the algorithm continues its execution.

In the second step, the algorithm translates the SDs with annotated state vectors, produced in the first step, into a Finite State Machine (FSM) for each individual SD, one for each instance in the SD. The algorithm assumes that messages directed away from an instance are actions, and messages directed toward an instance are events. A loop is detected under two conditions: If the state vector after the current message is the same as an existing state vector; and, if the message is state-changing which means the current state vector differs from previous state vector. In the last step, the algorithm merges different FSMs generated from different SDs by identifying similar states. The algorithm considers two states to be similar if they have the same state vector and they have at least one common incoming transition. The algorithm merges similar nodes by adding empty transitions between them. It uses a standard finite automata algorithm to remove these empty transitions.

4.1.1 Remarks on Whittle and Schumann

Whittle and Schumann \[17\] propose a technique that converts scenarios represented in UML sequence diagrams (SD) into statecharts and Unified Modeling Lan-
guage (UML) class diagrams. They make use of a state vector, which contains the values of system variables, to introduce a hierarchy by merging similar states to simplify the generated statecharts. However, they do not consider overlapping scenarios. Also, they do not address alternative alphabets because partial behavioral models are not considered in their technique.

4.2 Uchitel et al. (2003)

The inputs to Uchitel et al. (2003) algorithm [21]’s solution are a set of scenarios described in basic Message Sequence Charts (bMSCs), a high Message Sequence Charts (hMSCs) diagram 1, and labels, provided by the end user, which determine similar content inside the scenarios. The intermediate output of this approach is a local Finite State Process (FSP) which is used to produce a behavioral model represented as an LTS model.

This approach consists of five steps. In the first, the algorithm adds the label “B_MSC_name” at the top and “E_MSC_name” at the bottom of each instance. In the second, it constructs a relation which includes the top and bottom labels (continuation relation). This procedure uses hMSC diagrams and labels created in the first step in addition to the initial state. In the third, the algorithm divides each instance into sub-instances depending on the internal labels. Each sub-instance starts and ends with a label. In the last step, the algorithm combines the sub-instances, depending on the continuation relation and similar labels. In the fifth step, the algorithm translates instances into local Finite State Processes (FSP) as follows. It assigns to the first and the last label to the processes names and, sequence events to the process behavior. The generated FSP is converted into a behavioral model, represented as LTS, using a tool called LTSA.

1hMSC is a graph which describes how the bMSCs are related
4.2.1 Remarks on Uchitel et al.(2003)

Uchitel et al. [21] describe another algorithm to convert scenarios represented by MSC and hMSC diagrams into Finite Sequential Processes (FSP) an algebraic notation used to describe Labeled Transition System (LTS) models. This approach uses the LTSA tool to convert an FSP representation to a LTS model. However, their solution assumes there are no scenarios which overlap. Their approach, also, does not deal with partial behavioral models and alternative alphabets.

4.3 Damas et al.

The inputs to Damas et al. [18] are set of positive and negative scenarios described in MSCs, and a set of fluents provided by the system analyst. The output of this approach is a behavioral model, represented as an LTS model, decorated with state invariants. This approach consists of the three steps described as follows.

In the first, the algorithm creates a behavioral model from the positive and negative scenarios. The algorithm starts by creating an initial LTS model from the positive scenarios. Then, it examines nodes in ascending lexicographical order, in order to select two compatible states that can be merged. The algorithm then merges the selected states. If the resulting behavioral model is nondeterministic, its nondeterministic states that are merged. Subsequently, the algorithm checks to ensure the generated behavioral model, does not violate any of the negative scenarios. In the last step, the algorithm generates a new scenario and asks the end-user if it is positive or negative. The new scenario consists of two parts: the first is already accepted by the behavioral model and the second part is the new behavior introduced due to the merging of the nondeterministic states recursively.

In the second step, the algorithm converts the behavioral model for each instance
The conversion is done using the following three steps:

- If the event in the behavioral model is not related to the agent (it is not an incoming or outgoing arrow) then it is replaced by the empty transition and the event is eliminated.

- If the resulting behavioral model was a Non-deterministic Finite Automata (NFA), then the algorithm converts it to Deterministic Finite Automata (DFA).

- The resulting behavioral model is minimized.

In the third step, the algorithm *decorates* the behavioral model, by adding the values of fluents in each state. The goal of this phase is to assign values to the given set of fluents in each state in the behavioral model. These values are used for validation of the behavioral model by the system analyst or system engineer. The algorithm uses a four-value logic with *true*, *false*, *top*, and *bottom* as its states. *True* and *false* are used for usual meaning. *Top* means the value of the fluent is true in some paths and false in the others. *Bottom* means the value of the fluent is unknown. The algorithm introduces a function called a supremum function and is denoted by *sup*. This function returns its value by getting the higher value using the lattice in Figure 4-1. In this step, the algorithm starts by assigning a *bottom* value to the fluent in each state except the initial state. It assigns the initial value of the fluent to the initial state. Then starting from the initial state, the algorithm propagates the value

![Figure 4-1: Four values function.](image-url)
of the fluent in the current state to its neighbors states using the supremum function and the transition between the current state and its neighbor states.

### 4.3.1 Remarks on Damas et al.

Damas et al. [18] propose a technique to synthesize a behavioral model, decorated with state invariants, from a set of scenarios and set of properties expressed as a set of *fluents*. The algorithm does not create a partial behavioral model; therefore alternative alphabets are not addressed. The algorithm merges similar states, after the creation of the behavioral model, but it assumes that all scenarios start from the same initial state. As a consequence, not all types of overlapping scenario are covered.

### 4.4 Uchitel et al. (2009)

The inputs for Uchitel et al. (2009) [14] algorithm are a set of scenarios described as bMSCs, a hMSC diagram, and a set of *safety properties* described using FLTL formalism. The output of this approach is a partial behavioral model, represented as MTS model. It consists of the following steps.

In the first, the algorithm select an existing approach to convert the given set of scenarios into behavioral model represented as LTS model. Then, the algorithm adds a sink state \( SN \). Next, the algorithm adds the transition \( s \xrightarrow{\text{act-} \text{trans}(s)} SN \) to the behavioral model for each state \( s \). In this transition, \( \text{act} \) is the set of all transitions, and \( \text{trans}(s) \) is the outgoing transitions from state \( s \).

In the second step, for each safety property, \( \phi \), the algorithm creates its behavioral model. Then the algorithm removes all transitions not corresponding to an infinite trace, and remove all states that are unreachable from the initial state. Next, the algorithm converts the outgoing transitions for each state that has more than one outgoing transition to “maybe” transitions.
In the third step, the algorithm merges the model produced in the first with the model produced by the second using the merge operator “+”\(^2\).

### 4.4.1 Remarks on Uchitel et al. (2009)

Uchitel et al. [14] describe an enhanced technique over [21]. This technique synthesizes a partial behavioral model, represented in Modal Transition System (MTS), from a set of scenarios and a set of properties. This approach assumes there is already an existing algorithm which converts a set of scenarios to a behavioral model. Therefore, no mention to overlapping scenarios is included. Also, the authors state clearly that this approach assumes that all scenarios and properties have the same alphabet and that the approach would require an extension to deal with alternative alphabets.

### 4.5 Krka et al.

The inputs for Krka et al. [13] algorithm are a set of scenarios captured as UML sequence diagrams, and a set of properties captured as OCL constraints. The output of this approach is a set of component-level partial behavioral models, represented as MTS model. This approach consists of four phases.

In the first phase, the algorithm extracts constraints on the level of the component instead of the whole system. It starts by identifying both the provided and expected operations for each component. Then from these operations, it identifies the domain and scoped variables. After that, it processes the system constraints one by one and tailors them to be at the component level.

In the second phase, the algorithm generates an initial partial behavioral model, represented as MTS model, using the component-level constraints generated in the

\(^2\)The merge operator will be out of the scope of this work.
first phase. The initial state is identified either using the system requirement or by
the system engineer. Then, it expands the partial behavioral model, represented as
MTS model, by adding new transitions and new states. A new transition \((s \xrightarrow{OP} m s')\)
is added if OPs pre and post-condition are satisfied in \(s\) and \(s'\), respectively. A new
state \(s\) is added if is a new transition \((s \xrightarrow{OP} m s')\) and “\(s\)” is not already defined in
the partial behavioral model.

In the third phase, sequence diagrams are annotated for each component with
values from component-level constraints. The algorithm adds the values of significant
domain variables depending on the component-level constraints.

In the forth phase, the algorithm starts by identifying the initial state, launching
\(MTS\) state, by identifying the compatible state between annotated SD, generated in
phase(2), and initial MTS, generated in phase(3). Then, starting from the initial
state, launching \(MTS\), the algorithm traverses the initial MTS using the annotated
SD sequence diagram. The algorithm converts the traversed transition into a required
transition if it is a potential transition. In some cases, when the transition is a self-
transition (transition back to self) or backward-transition, the algorithms adds new
states to avoid over-specified specification.

4.5.1 Remarks on Krka et al.

Krka et al. [13] propose a technique to synthesizes a partial behavioral model,
expressed as an MTS model, from a set of scenarios and a set of properties, expressed
as Object Constraint Language (OCL) constraints. Krka et al. state clearly that their
technique does not take into account alternative alphabets or overlapping scenarios.
4.6 Summary

The listed works introduce algorithms to synthesize behavioral models from user requirements expressed as a set of scenarios and a set of properties. However, none of these algorithms and techniques consider scenarios with alternative alphabets [14]. Some algorithms such as [18] consider overlapping scenarios but do not consider all the possible types of overlapping scenarios [21].
Chapter 5

Related Algorithms

A Synthesis algorithm, in the context of this work, is an algorithm that converts a set of scenarios or a set of properties into a behavioral model. Then, it converts the resulting behavioral model into a partial behavioral model. Section 5.1 and Section 5.2 explain the algorithms used to synthesize both the behavioral model and the partial behavioral model from scenarios and properties, respectively.

5.1 Mapping Scenarios to Behavioral Model

The process of mapping scenarios to partial behavioral model is done by the creation of the behavioral model for the scenarios followed by the conversion of the behavioral model for the scenarios into a partial behavioral model. The creation and conversion processes are discussed in the subsections that follow.

5.1.1 Creating the Behavioral Model of Scenarios

In this subsection, we will describe the algorithm developed by Uchitel et al. [21] to convert scenarios into a behavioral model.

This algorithm in [21] consists of five steps. In the first step, the algorithm adds the label $B_{MSC\_name}$ at the top and $E_{MSC\_name}$ at the bottom of each instance.
In the second step, the algorithm constructs a relation that includes the top and bottom labels (continuation relation) using hMSC diagrams and labels created in the first step in addition to initial state. In the third step, it divides each instance into sub-instances depending on the internal labels. Each sub-instance starts and ends with a label. In the forth step, it combines the sub-instances, depending on the continuation relation and similar labels. In the fifth step, the algorithm translates instances into local Finite State Processes (FSP), by assigning the first label as the process name, the final label as the another process name, and the other sequence events as the process behavior. The generated FSP is converted into a behavioral model, represented as LTS, using a tool called LTSA tool.

Although this algorithm is rigid, we propose a more straightforward algorithm, Algorithm 5-1. The following reasons justify this decision. (1) It avoids using labels which are not suitable in practical situations [18], (2) it avoids using other notations, namely FSP processes, that are not relevant directly to the problem, and it does not require the LTSA tool to create the behavioral model from FSP processes.

**Input:** hMSC, set of scenarios S

**Output:** The behavioral model of each instance

1: for all instance $inst_i \in$ instance-set do
2: \hspace{1cm} behavioral-model$_{inst_i} = \epsilon$
3: for all edge $(u, v, scenario-sc_i) \in$ hMSC do
4: \hspace{1cm} newEdge = \epsilon
5: \hspace{1cm} for all transition$_j \in sc_i$ such that sender = $inst_i$ or receiver = $inst_i$ do
6: \hspace{2cm} newEdge = newEdge + transition$_j$
7: \hspace{1cm} end for
8: \hspace{1cm} add newEdge to behavioral-model-$inst_i$
9: end for
10: end for

Figure 5-1: Create behavioral model for each instance.

For example, suppose we have the set of scenarios shown in Figure 5-2. We apply Algorithm 5-1 to create the behavioral model for the instances (User, Control
unit, Alarm unit, and Timer unit), Algorithm 5-1 starts by converting interactions in scenarios into transitions in the behavioral model. For example, the behavioral model of User instance, Control Unit instance, Alarm Unit instance, and Timer Unit instance are shown in the Figure 5-3. The second step is to compose scenario instances in one LTS model. This composition can be done by parallel composition of the behavioral models of all instances [21].

5.1.2 Convert The Behavioral Model of Scenarios into Partial Behavioral Model

To convert a behavioral model of scenarios into a partial behavioral model, the algorithm performs the following steps. It starts by adding a new state, the sink state. Then, for each state, \( s \), in the behavioral model if there is no outgoing transition \( t' \) from this state, the algorithm adds a maybe transition \( t' \) from state \( s \) to state sink [14].

For example, to convert the behavioral model shown in Figure 5-5(a), to a partial behavioral model, the following steps are applied. First, the algorithm adds a sink state, state -1, to Figure 5-5(a). Then, for state 0, the algorithm adds transitions \( state_0 \xrightarrow{Act-\{}\{\text{start ringing}\}\} state_{-1} \). Next, for state 1, the algorithm adds transitions \( state_1 \xrightarrow{Act-\{}\{\text{stop ringing}\}\} state_{-1} \). \( Act-\{}\{\text{stop ringing}\}\) from state 1 to state -1 (sink). Finally, for state -1, the algorithm adds transitions \( state_{-1} \xrightarrow{Act}\} state_{-1} \).

Figure 5-5(c) shows the partial behavioral model of the behavioral model in Figure 5-5(a).

5.2 Mapping Properties to Behavioral Model

The process of mapping properties into partial behavioral model is done by the creation of the behavioral model for the properties followed by the conversion of the
(a) Scenario Sc1.

(b) Scenario Sc2.

(c) Scenario Sc3.

(d) Scenario Sc4.

(e) Diagram showing the order of execution of scenarios.

Figure 5-2: Set of scenarios that illustrate the behavior of an alarm in a clock execution.
Figure 5-3: The behavioral models of the instances in Figure 5-2

Figure 5-4: The composition of the instance behavioral models shown in Figure 5-2
behavioral model for the properties into a partial behavioral model. The creation and conversion processes are discussed in the subsections that follow.

5.2.1 Creating The Behavioral Model of Properties

A system's property can be described as a set of traces that satisfy this property [29]. To convert a system's property, described as an FLTL formalism, to behavioral model represented as an LTS model, the following needs to be performed [36]: (1) rewrite the system’s properties in a special format, in which only specific operators are used, and (2) create a Buchi automaton for these properties. We will refer to step (1) and step (2) as rewriting properties and Buchi automaton creation, respectively. Both the rewriting properties and Buchi automaton creation steps are described as follows.

Rewriting property

Rewriting property is done through two steps. We start by converting the property into a format where negation applies only to events or fluents by applying the following rules [36]:

\[-(\alpha U \beta) \equiv ((\neg \alpha) R (\neg \beta))\]

\[-(\alpha R \beta) \equiv ((\neg \text{alpha}) U (\neg \beta))\]

\[-X \alpha \equiv X \neg \alpha\]
\[ \neg \Box \alpha \equiv \Diamond \neg \alpha \]
\[ \neg \Diamond \alpha \equiv \Box \neg \alpha \]

Next, we eliminate both \( \Box \), \( \Diamond \) operators by applying the following rules [36]:
\[ \Box \alpha \equiv (False \alpha) \]
\[ \Diamond \alpha \equiv (True \alpha) \]

**Buchi automaton creation**

The behavioral model of a property is characterized by the Buchi automaton representation of this property. Such an automaton is constructed by creating a set of states and a set of transitions that represent this property. For example, the behavioral model for the property \( a \lor b \) is shown in Figure 5-6(a), which is interpreted as from the starting state the model can go to the final state by either transitions \( a \) or \( b \). Figure 5-6(b) illustrates the behavioral model for the property \( c \land d \) which means that the transition from the initial state to the final state is accomplished by satisfying both \( c \) and \( d \).

Figure 5-6: (a) The behavioral model of \( a \lor b \) property and (b) The behavioral model of \( a \land b \) property.

One algorithm, that converts properties to a behavioral model, is the LTL2Buchi algorithm [36]. The main idea behind this algorithm is the expansion of states such that it represents the formula. The detailed description of this algorithm can be obtained in [36]. We omit these details for the sake of simplicity.

Two tools which implement the LTL2Buchi algorithm are, the Java Path Finder from NASA [36] and Modal Transition System Analyser (MTSA) [14]. However,
such tools give different models for the same property. In this study, we will use the implementation of MTSA tool [14].

Uchitel et al.’s algorithm\textsuperscript{1} [14], shown in Figure 5-7, creates a behavioral model from properties.

\begin{align*}
1: & \textbf{for all} \ \text{properties } \varphi \ \textbf{do} \\
2: & \quad \text{Construct a Buchi automaton.} \\
3: & \quad \text{Remove transitions that correspond to a finite trace.} \\
4: & \quad \text{Remove all unreachable states.} \\
5: & \textbf{end for}
\end{align*}

Figure 5-7: Creation of a behavioral model for properties without considering alternative alphabets.

\subsection{5.2.2 Convert The Behavioral Model of Properties Into Partial Behavioral Model}

To convert a behavioral model, represented as an LTS model, to a partial behavioral model, represented as an MTS model, we convert all outgoing transitions to \textit{may be} transitions\textsuperscript{2} for any state that has more than one outgoing transition.

\begin{align*}
1: & \textbf{for all} \ \text{node } n \in \text{nodeset } \textbf{do} \\
2: & \quad \textbf{if} \ \text{number of outgoing transitions from } n > 1 \ \textbf{then} \\
3: & \quad \quad \text{convert all the outgoing transitions } t \text{ to } \textit{maybe} \text{ transition.} \\
4: & \quad \textbf{end if} \\
5: & \textbf{end for}
\end{align*}

Figure 5-8: Conversion of a behavioral model of properties, represented as an LTS model into a partial behavioral model, represented as an MTS model.

Uchitel et al.’s algorithm [14], shown in Figure 5-8, converts the behavioral model of properties into partial behavioral model of properties without considering alternative alphabets.

\footnote{\textsuperscript{1}It uses LTL2BUCHI and does not consider alternative alphabets.} \\
\footnote{\textsuperscript{2}A \textit{may be} transition can be allowed or disallowed, but the decision about that was not yet taken.}
Chapter 6

Detecting and Decomposing Overlapping Scenarios Using Pattern Matching

6.1 Introduction

Scenarios are written descriptions used to organize the information generated by requirements analysis [8]. These descriptions are typically in story form giving a sequence of events. Representing user requirements as a set of scenarios is an established activity in software engineering [3, 4] and its use makes the communication between end users and software developers more efficient [9]. A behavioral model is a formalism that describes the behavior of the system using a sequence of state transitions [9]. It also provides a graphical representation for the new system to be developed [9]. Scenarios can be used to create a behavioral model for the system to be developed [21] and such model can be used to validate the user requirements [21, 37]. Similarly, scenarios may also be used to validate both system analysis and design [38]. Overlapping scenarios can cause suboptimal behavioral models, therefore identifying and decomposing scenarios is important [21]. Each scenario contains a set of events and these
events have chronological order, which means they can be treated as a sequence [25]. This chapter proposes an extension to pattern matching technique [39–42] to detect overlapping scenarios.

![Figure 6-1: Two scenarios represented in the MSC notation.](image)

Message Sequence Charts (MSC) are sequence diagrams used to represent scenarios [11]. For example, Figure 6-1 shows two MSCs. In order to explain our approach, we will record the MSC information in a structure. MSC name, MSC starting point (front join point), MSC end point (rear join point), and the sequence of events are the components of this structure. Each event contains the sender entity, the receiver entity, and the message (action). Table 6.1 records the information of MSCs shown in Figure 6-1.

The chapter is structured as follows. In section 6.2, we describe a proposed solution for the overlapping scenarios problem. In section 6.3, we discuss pattern matching and show how the overlapping scenarios can be detected using this approach. In section 6.4, we describe how to decompose overlapping scenarios and how to create a new scenario for the overlapping part. Conclusions are given in section 6.5.
6.2 Decomposing Overlapping Scenarios

Overlapping scenarios contain shared parts. These parts can result in a behavioral model with redundant states. To address this problem we introduce a way to eliminate such redundancies. The solution is composed of two parts or steps. Initially it decomposes the overlapping scenarios into smaller scenarios without shared parts. This step eliminates all redundant states that will appear in the resulting behavioral model and it will be referred to as decomposition. The step that follows creates the behavioral model using existing synthesis algorithms. For this reason it will be referred to as synthesis. A detailed description of these steps is discussed as follows.

6.2.1 Decomposition

We propose an algorithm that employs pattern matching \cite{39–42} to find similar parts in scenarios. The algorithm decomposes each scenario into two or more parts depending on the position of overlapping. To keep the proper sequence\footnote{The order of events in scenarios.} of the decomposed scenarios, we introduce join points which are links that relate the end of one scenario to the beginning of another. The decomposition involves the following activities:

1. Find the overlapping scenarios,
2. Decompose the overlapping scenarios, and
3. Add join points to keep the proper sequence of the decomposed scenarios.

Assume, for instance, scenarios Sc1 and Sc2, as shown in Table 6.1. Also assume that the pattern matching algorithm finds that Sc1 and Sc2 have an overlapping part, “(P, A, Q)”, and non-overlapping parts “(R, B, Q)” and “(Q, C, R)”. The
algorithm then decomposes scenario Sc1 into two scenarios Sc1’ and Sc3. Similarly, it decomposes Sc2 into two scenarios Sc2’ and Sc3. Also the algorithm introduces the join point J1 at the end of new scenario Sc3 and join point J2 at the beginning of both scenarios Sc1’ and Sc2’ as shown in Table 6.2.

The output of the decomposition step is a set of scenarios, therefore it can be used as an input for the synthesis step.

### 6.2.2 Synthesis (Behavioral Model Creation)

After the scenario decomposition existing synthesis algorithms can be used to create the behavioral model. Details of such process can be obtained in [14,17,18,21].

### 6.2.3 Techniques to Identify Overlapping Scenarios

Two techniques can be used to detect the overlapping scenarios. They are pattern matching [39–41] and sequence comparisons [43–45]. Both approaches may be used to find overlapping scenarios, because they have similar input and output. The input of both pattern matching and sequence comparisons are ordered sequences of items (i.e.
Table 6.2: Set of scenarios after decomposition.

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc1'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top join point</td>
<td>J2</td>
</tr>
<tr>
<td>Rear join point</td>
<td>null</td>
</tr>
<tr>
<td>Events</td>
<td>(R, B, Q)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc2'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top join point</td>
<td>J2</td>
</tr>
<tr>
<td>Rear join point</td>
<td>null</td>
</tr>
<tr>
<td>Events</td>
<td>(Q, C, R)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top join point</td>
<td>null</td>
</tr>
<tr>
<td>Rear join point</td>
<td>J1</td>
</tr>
<tr>
<td>Events</td>
<td>(P, A, Q)</td>
</tr>
</tbody>
</table>

strings). Similarly, the input for finding overlapping scenarios procedures are ordered sequences of events. The output of both pattern matching and sequence comparisons techniques are either no match or the positions of the match. Similarly, the output of finding overlapping scenarios procedures is no overlapping or the position of the overlapping. In this work, we use pattern matching.

6.2.4 The Proposed Approach to Detect and Decompose Scenarios

Our approach consists of two main stages. In the first stage, the algorithm finds all possible overlaps between two scenarios and record these results in a score matrix. Then, it uses backtracking to find all possible overlapping parts between the two scenarios. Next, it selects one part of the set of all possible overlaps. Finally, it finds other scenarios which share the same overlapping part.

In the second stage, the algorithm decomposes the overlapping scenarios and cre-
ates a new scenario for the overlapping parts. To keep the sequence of scenarios, the algorithm adds *join points*[^2]. Figure 6-2 shows the main activities of the proposed approach.

![Diagram](image)

**Figure 6-2:** The proposed algorithm consists of two stages. In the first stage, which consists of four steps, it detects the overlapping scenarios using pattern matching. In the second stage, which consists of two steps, it decomposes the overlapping scenarios, and creates a new scenario for the overlapping part.

### 6.3 Detecting Overlapping Scenarios Using Pattern Matching

Finding the longest common sequence (LCS) is a well-known application of pattern matching [39]. Implementing such an algorithm can be done using different

[^2]: The join points we use here are similar to the continuation relation used in [21]. However, in our approach, the join points are dynamic, which means each iteration can change the status of the join points. On the other hand, the continuation relation is static, which means it contains only the initial state of the scenarios.
approaches, such as brute force or dynamic programming. To find an overlap between two scenarios, say, for instance, Sc1 and Sc2, a brute force algorithm examines all events in Sc1 and determines the existence of overlaps with the events in scenario Sc2. However, the time complexity of this algorithm is \(O(2^n)\), because a scenario Sc1 with \(n\) events has \(2^n\) events combinations [39]. On the other hand, the time complexity of dynamic programming is \(O(mn)\) [39]. For this reason the methods that attempt to find the LCS are usually based on dynamic programming technique [42].

The solution we propose to detect overlapping scenarios is composed of four steps and it is based on dynamic programming. In the first step, a score matrix is created. While, in the second, backtracking is applied to find the LCS. In the third step, an overlap is selected and in the fourth, the algorithm searches for other scenarios that have the same overlap. These steps are described as follows.

### 6.3.1 Creating the Score Matrix \(T\)

For any given two scenarios, say for instance S and U we use dynamic programming to create the score matrix \(T[0..n, 0..m]\), where \(n\) is the number of events in scenario S and \(m\) is the number of events in scenario U. The entry \(T[i,j]\) represents the number of overlapping events between \(S[1..i]\) and \(U[1..j]\). We use the following definition to compute \(T[i,j]\) [42].

\[
T(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
T[i-1, j-1]+1 & \text{if } S.event(i) = U.event(j) \\
\max(T[i,j-1], T[i-1,j]) & \text{if } S.event(i) \neq U.event(j)
\end{cases}
\]

The algorithm shown in Figure 6-3 illustrates the required steps necessary for computing matrix \(T\) entries [39]. For instance, Table 6.6 presents the score matrix of scenarios Sc1 and Sc2 which are shown in Tables 6.3 and 6.4.
Input: Sc1, Sc2: scenario
Output: Score matrix T
1: \( n = \) number of events in Sc1
2: \( m = \) number of events in Sc2
3: for \( i = 0 \) to \( n \) do
4: \( T[i, 0] = 0 \)
5: end for
6: for \( j = 0 \) to \( m \) do
7: \( T[0, j] = 0 \)
8: end for
9: for \( i = 1 \) to \( n \) do
10: for \( j = 1 \) to \( m \) do
11: if \( SC1.events[i] = SC2.events[j] \) then
12: \( T[i + 1, j + 1] = T[i, j] + 1 \)
13: else
14: \( T[i + 1, j + 1] = \max(T[i + 1, j], T[i, j + 1]) \)
15: end if
16: end for
17: end for

Figure 6-3: Creates the score matrix T for two scenarios Sc1 and Sc2.

<table>
<thead>
<tr>
<th>Table 6.3: Scenario Sc1 before decomposition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario name</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Front join point</td>
</tr>
<tr>
<td>Rear join point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Events</th>
<th>Sc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( user, enter password, ATM)</td>
</tr>
<tr>
<td>( 1 )</td>
<td>(ATM, verify account, consortium)</td>
</tr>
<tr>
<td>( 2 )</td>
<td>(consortium, verify card with bank, bank)</td>
</tr>
<tr>
<td>( 3 )</td>
<td>(bank, bad bank password, consortium)</td>
</tr>
<tr>
<td>( 4 )</td>
<td>(consortium, bad password, ATM)</td>
</tr>
<tr>
<td>( 5 )</td>
<td>(ATM, request password, user)</td>
</tr>
</tbody>
</table>
Table 6.4: Scenario Sc2 before decomposition.

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front join point</td>
<td>0</td>
</tr>
<tr>
<td>Rear join point</td>
<td>1</td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>(user, enter password, ATM)</td>
</tr>
<tr>
<td>(1)</td>
<td>(ATM, verify account, consortium)</td>
</tr>
<tr>
<td>(2)</td>
<td>(user, cancel, ATM)</td>
</tr>
<tr>
<td>(3)</td>
<td>(ATM, cancelled message, user)</td>
</tr>
<tr>
<td>(4)</td>
<td>(ATM, eject card, user)</td>
</tr>
<tr>
<td>(5)</td>
<td>(ATM, request take card, user)</td>
</tr>
<tr>
<td>(6)</td>
<td>(user, take card, ATM)</td>
</tr>
</tbody>
</table>

Table 6.5: Scenario Sc3 before decomposition.

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front join point</td>
<td>0</td>
</tr>
<tr>
<td>Rear join point</td>
<td>1</td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>(0)</td>
<td>(user, enter password, ATM)</td>
</tr>
<tr>
<td>(1)</td>
<td>(ATM, verify account, consortium)</td>
</tr>
<tr>
<td>(2)</td>
<td>(consortium, verify card with bank, bank)</td>
</tr>
<tr>
<td>(3)</td>
<td>(bank, bad bank account, consortium)</td>
</tr>
<tr>
<td>(4)</td>
<td>(consortium, bad account, ATM)</td>
</tr>
<tr>
<td>(5)</td>
<td>(ATM, bad account message, user)</td>
</tr>
<tr>
<td>(6)</td>
<td>(ATM, print receipt, user)</td>
</tr>
<tr>
<td>(7)</td>
<td>(ATM, eject card, user)</td>
</tr>
<tr>
<td>(8)</td>
<td>(ATM, request take card, user)</td>
</tr>
<tr>
<td>(9)</td>
<td>(user, take card, ATM)</td>
</tr>
</tbody>
</table>
Table 6.6: Score matrix of scenarios Sc1 and Sc2 using pattern matching.

<table>
<thead>
<tr>
<th>sc2</th>
<th>(user, enter password, ATM)</th>
<th>(ATM, verify account, consortium)</th>
<th>(user, cancel, ATM)</th>
<th>(ATM, cancelled message, user)</th>
<th>(ATM, eject card, user)</th>
<th>(ATM, request take card, user)</th>
<th>(user, take card, ATM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(user, enter password, ATM)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(ATM, verify account, consortium)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(consortium, verify card with bank, bank)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(bank, bad bank password, consortium)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(consortium, bad password, ATM)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

6.3.2 Backtracking Entires in the Score Matrix T to Find the Overlapping Parts

After the generation of the score matrix $T$, we extract the overlapping patterns by backtracking the matrix $T$ from the end point $T[n, m]$ to the starting point $T[0, 0]$. The number of overlapping events between the two scenarios will be the largest entry in the score matrix [42]. The algorithm shown in Figure 6-4 illustrates the details involved in tracing entires in the score matrix $T$ to find the overlapping events between the two scenarios [39].

To extract the overlapping events between the two scenarios sc1 and sc2 shown in Tables 6.3 and 6.4, we use the algorithm presented in Figure 6-4 which traces their score matrix, starting from the end point as illustrated in Table 6.7. Two components result from this step. The first is the overlapping events shared between the two scenarios and the second is an index which determines the position of the overlapping
Input: $Sc_1$, $Sc_2$: scenario, score matrix $T$
Output: $OverlappingParts$
1: $n = $ number of events in $Sc_1$
2: $m = $ number of events in $Sc_2$
3: $index = T[n, m]$
4: while $i > 0$ and $j > 0$ do
5: \[ \text{if } Sc_1.events[i - 1] = Sc_2.events[j - 1] \text{ then} \]
6: \[ OverlappingParts.events[index] = Sc_1.events[i - 1] \]
7: \[ OverlappingParts.pos1[index] = i - 1 \]
8: \[ OverlappingParts.pos2[index] = j - 1 \]
9: $index = index - 1$
10: $i = i - 1$
11: $j = j - 1$
12: else if $T[i - 1, j] > T[i, j - 1]$ then
13: $i = i - 1$
14: else
15: $j = j - 1$
16: end if
17: end while

Figure 6-4: Extracts the overlapping events between scenarios $Sc_1$ and $Sc_2$ by tracing their score matrix.

Table 6.7: Tracing the score matrix of scenarios $Sc_1$ and $Sc_2$ to extract the overlapping part.
events in each scenario. Table 6.8 presents the overlapping events between scenarios sc1 and sc2 shown in Tables 6.3 and 6.4. Table 6.9 shows the overlapping scenario set. This set contains the scenario name and the starting position of overlapping in the scenario events.

Table 6.8: Overlapping events between two scenarios, Sc1 and Sc2, using pattern matching.

<table>
<thead>
<tr>
<th>index</th>
<th>event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(user, enter password, ATM)</td>
</tr>
<tr>
<td>1</td>
<td>(ATM, verify account, consortium)</td>
</tr>
</tbody>
</table>

Table 6.9: Overlapping scenarios set for scenarios Sc1 and Sc2 using pattern matching.

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sc1</td>
<td>0</td>
</tr>
<tr>
<td>Sc2</td>
<td>0</td>
</tr>
</tbody>
</table>

6.3.3 Overlapping Part Selection

The overlapping events that were extracted in the previous step can be discontinuous. This means they are not adjacent in the original scenario. Since a scenario consists of a sequence of events, we need to select one continuous overlapping part. We will use a heuristic that selects the largest overlapping part from the set of all overlapping parts. This means the overlapping part that contains the largest number of events. The algorithm shown in Figure 6-5 uses the largest overlapping part heuristic. It returns the overlapping part that has the maximum number of consecutive events. In our running example, Table 6.8 contains only one continuous overlapping part, therefore, the output of this step will be the same as Table 6.8.
Input: OverlappingParts, score matrix T
Output: maximum_OverlappingPart

1: count = 0
2: tempCount = 0
3: index = 0
4: tempIndex = 0
5: index = T[n, m]
6: for i = 0 to index do
7:     if OverlappingParts.pos1[i] + 1 = OverlappingParts.pos1[i + 1] and OverlappingParts.pos2[i] + 1 = OverlappingParts.pos2[i + 1] then
8:         tempCount = tempCount + 1
9:     else
10:        if tempCount > count then
11:            count = tempCount
12:            index = tempIndex
13:        end if
14:        tempIndex = i + 1
15:        tempCount = 0
16:    end if
17: end for
18: return OverlappingParts.events[index, index + count]

Figure 6-5: Finding the maximum overlapping part from the set of the overlapping parts.
6.3.4 Find Scenarios That Share The Same Overlapping Part

After we determine the overlapping part between two scenarios, we need to check if other scenarios have the same overlapping part. This can be done by comparing the overlapping part with other scenarios for an exact match. Resulting matches are added to the overlapping scenarios set. The algorithm shown in Figure 6-6 checks if the overlapping part exists in a scenario. If that is the case it returns the event number in which the overlapping starts. In our running example, the algorithm shown in Figure 6-6 determines that scenario sc3, presented in Table 6.5, has the same overlapping part and returns the position of the beginning of the overlapping. The resulting overlapping scenarios set is illustrated in Table 6.10.

Input: Sc1: scenario, OV: overlapping part
Output: an index where Sc1 completely matches OV or null otherwise

1: n = number of events in Sc1
2: m = number of events in OV
3: for s = 0 to n – m do
4:   if the events from OV.events[1] to OV.events[m] matches the events from SC1.events[s + 1] to event SC1.events[s + m] then
5:     return OV matches SC1 at index = s + 1
6:   end if
7: end for
8: return null

Figure 6-6: Find other scenarios that have the same overlapping part.

Table 6.10: Overlapping scenarios set after finding other scenarios that share the same overlapping parts.

<table>
<thead>
<tr>
<th>Scenario Name</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sc1</td>
<td>0</td>
</tr>
<tr>
<td>Sc2</td>
<td>0</td>
</tr>
<tr>
<td>Sc3</td>
<td>0</td>
</tr>
</tbody>
</table>
6.4 Decomposing Scenarios and Creating a New Scenario For the Overlapping

As we have seen in the previous section, pattern matching and LCS can be extended to find the overlapping events between two scenarios. In this section we introduce an algorithm to eliminate the overlaps between scenarios. This algorithm identifies the overlapping parts between scenarios and it creates a new scenario for the overlapping events. It also decomposes the scenarios that have overlapping events. The algorithm shown in Figure 6-7 provides the details of this method. In the sections that follow we describe the main activities in the second stage that includes creating a new scenario from the overlapping part and decomposing scenarios in the overlapping scenarios set.

6.4.1 Creating a New Scenario From the Overlapping Events

In this step we create a scenario for the overlapping events that are shared between scenarios in the scenarios overlapping set. This can be done by creating a scenario that contains the shared events. But, before that we need to check if this scenario already exists. For this case there will be no need to create a new scenario. For the current example, we still need to add a set of join points to keep the correct sequence of events in both the new created scenario and in the decomposed scenarios. The algorithm in Figure 6-9 illustrates the steps related to the creation of new scenarios depending of events in the overlapping part. In our running example, this step will create scenario NewSc shown in Table 6.11.
Input: set of scenarios $S$
Output: set of scenarios $S$
1: for all scenario $Sc_i \in$ scenarios set $S$ do
2: {examine each scenario in the scenarios set $S$}
3: morePossibleOverlapping = true
4: while morePossibleOverlapping do
5:   for all $(Sc_j \in$ scenarios set $S) \land (j > i)$ do
6:     {Check if there is a match between scenarios $Sc_i$ and another scenario $Sc_j$}
7:     OverlappingPart = Find-a-match $(Sc_j, Sc_i)$
8:     if $OverlappingPart \neq$ null then
9:       add $Sc_j$ to overlapped-scenarios-set
10:      $OverlappingPart = \text{maximum}(\text{overlappingPart-set})$
11:     for all scenario $Sc_k \in$ scenarios set $S - (Sc_i, Sc_j)$ do
12:       {if we find a match between a pair of scenarios $Sc_i$ and $Sc_j$, then we try to find if another scenario have the same overlapping part}
13:       if Find-exact-match $(OverlappingPart, Sc_k) \neq$ null then
14:         add $Sc_k$ to overlapped-scenarios-set
15:     end if
16:   end for
17:   decompose-scenario (overlapping-scenarios-set)
18:   {create a new scenario for the overlapping part and decompose the scenarios in scenarios set}
19: else
20:   more-possible-overlapping = false
21:   {there is no more overlapping in this scenario move to next one}
22: end if
23: end for
24: end while
25: end for

Figure 6-7: The proposed algorithm for the identification and decomposition of overlapping scenarios.
Figure 6-8: The flowchart of the proposed algorithm that decomposes a set of overlapping scenarios.

**Input:** $Sc$: scenario, $count$, $index$: number, $frontJP$, $rearJP$: joinpoints

**Output:** $newSc$: scenario

1. $newSc = createNewScenario()$
2. $newSc.frontJoinPoint = frontJP$
3. $newSc.rearJoinPoint = rearJP$
4. for $i = 0$ to $count$ do
5. \hspace{1em} $newSc.event[i] = Sc.event[i + index]$
6. end for
7. add the scenario $newSc$ to scenarios set $S$

Figure 6-9: Creates a new scenario by specifying the scenarios $Sc$ which has the overlapping part, the starting of the overlapping $index$ and $count$ the number of events in the overlapping part.
Table 6.11: Scenario Sc1 after decomposition.

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc1’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front join point</td>
<td>2</td>
</tr>
<tr>
<td>Rear join point</td>
<td>1</td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>(consortium, verify card with bank, bank)</td>
</tr>
<tr>
<td>( 1 )</td>
<td>(bank, bad bank password, consortium)</td>
</tr>
<tr>
<td>( 2 )</td>
<td>(consortium, bad password, ATM)</td>
</tr>
<tr>
<td>( 3 )</td>
<td>(ATM, request password, user)</td>
</tr>
</tbody>
</table>

Table 6.12: Scenario Sc2 after decomposition

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc2’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front join point</td>
<td>2</td>
</tr>
<tr>
<td>Rear join point</td>
<td>1</td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>(user, cancel, ATM)</td>
</tr>
<tr>
<td>( 1 )</td>
<td>(ATM, cancelled message, user)</td>
</tr>
<tr>
<td>( 2 )</td>
<td>(ATM, eject card, user)</td>
</tr>
<tr>
<td>( 3 )</td>
<td>(ATM, request take card, user)</td>
</tr>
<tr>
<td>( 4 )</td>
<td>(user, take card, ATM)</td>
</tr>
</tbody>
</table>
Table 6.13: Scenario Sc3 after decomposition

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>Sc3'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front join point</td>
<td>2</td>
</tr>
<tr>
<td>Rear join point</td>
<td>1</td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>(consortium, verify card with bank, bank)</td>
</tr>
<tr>
<td>( 1 )</td>
<td>(bank, bad bank account, consortium)</td>
</tr>
<tr>
<td>( 2 )</td>
<td>(consortium, bad account, ATM)</td>
</tr>
<tr>
<td>( 3 )</td>
<td>(ATM, bad account message, user)</td>
</tr>
<tr>
<td>( 4 )</td>
<td>(ATM, print receipt, user)</td>
</tr>
<tr>
<td>( 5 )</td>
<td>(ATM, eject card, user)</td>
</tr>
<tr>
<td>( 6 )</td>
<td>(ATM, request take card, user)</td>
</tr>
<tr>
<td>( 7 )</td>
<td>(user, take card, ATM)</td>
</tr>
</tbody>
</table>

Table 6.14: Scenario NewSc that is generated after decomposition

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>NewSc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front join point</td>
<td>0</td>
</tr>
<tr>
<td>Rear join point</td>
<td>2</td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>( 0 )</td>
<td>(user, enter password, ATM)</td>
</tr>
<tr>
<td>( 1 )</td>
<td>(ATM, verify account, consortium)</td>
</tr>
</tbody>
</table>
6.4.2 Decomposing Scenarios in the Overlapping Scenarios Set

The goal of this step is to decompose scenarios in the overlapping scenarios set. Decomposing a scenario depends on the position of the overlapping part in this scenario. The position of the overlapping part can be either

(a) at the beginning of the scenario, or

(b) at the end of the scenario, or

(c) in the middle of the scenario.

In both case(a) and case(b), the algorithm removes the overlapping part, renumbers the events in the scenario and adds one pair of join points as illustrated in Figure 6-10(a) and Figure 6-10(b). In case(c) the algorithm creates another scenario from the events beyond the overlapping part. It also removes the overlapping part, renumbers the events in the scenario, and adds two pairs of join points. Figure 6-10(c) illustrates this activity.

Assume, for instance, that \( m \) is the number of events in the scenario, \( count \) is the number of events in the overlapping part before decomposition and \( index \) indicates the starting position of the overlapping in the scenario. After decomposition, the number of events in the scenario will be \( m - count \) and the number of events in the new scenario will be \( count \) for both case(a) and case(b). In case(c), the number of events in the scenario will be \( index \). The number of events in the new scenarios will be \( count \) and \( m - count - index \). The algorithm in Figure 6-11 illustrates the detailed description of the scenario decomposition algorithm.

Before the decomposition of any scenario, we need to check if this decomposition produces an empty scenario. For such a case the scenario does not need to be decomposed.
In our running example, this step decomposes scenarios sc1, sc2, and sc3 and produces a new version of them as illustrated in Tables 6.11, 6.12, and 6.13.

Figure 6-10: In decomposing scenarios and adding join points, we have three cases: the overlapping part at the beginning of the scenario, the overlapping part at the end of the scenario, and the overlapping part in the middle of the scenario. Figures (a), (b), and (c) illustrate the different cases respectively.

6.5 Conclusion

The elimination of overlapping scenarios was approached by the use of pattern matching. The algorithms proposed structured the solution in two stages, the identification of the overlapping and the decomposition of the scenarios. Although the number of scenarios may increase, the overall number of events is minimized. The strategy proposed made possible the creation of behavioral models that are free from the redundancies caused by the presence of the overlapping scenarios.
Input: $S$: scenario set, $Sc$: scenario, $index$, $count$: number
Output: $S$: scenario set, $newSc$: scenario
1: $m =$ number of events in $Sc$
2: if $index = 0$ then
3:   { The first case, the overlapping part at the top of the scenario}
4:   $newJP = createNewJoinPoint$
5:   $newSc = create-new-scenario (Sc, index, count, Sc.frontJoinPoint, newJP)$
6:   $Sc.frontJoinPoint = newJP$
7:   for $i = 0$ to $m - count$ do
8:     $Sc.event[i] = Sc.event[i + index]$
9:   end for
10: the number of events in scenario $Sc = m - count$
11: else if $count + index = m$ then
12:   { The second case, the overlapping part at the bottom of the scenario }
13:   $newJP = createNewJoinPoint$
14:   $newSc = create-new-scenario(Sc, index, m, newJP, Sc.rearJoinPoint)$
15:   $Sc.rearJoinPoint = newJP$
16:   for $i = m - count$ to $m$ do
17:     remove $Sc.event[i]$
18:   end for
19: the number of events in scenario $sc = m - count$
20: else
21:   { The Third case, the overlapping part in the middle of the scenario }
22:   $newJP1 = createNewJoinPoint$
23:   $newJP2 = createNewJoinPoint$
24:   $newSc1 = create-new-scenario(Sc, index, index + count, newJP1, newJP1)$
25:   $newSc2 = create-new-scenario(Sc, index + count, m, newJP2, Sc.rearJoinPoint)$
26:   $Sc.rearJoinPoint = newJP1$
27:   for $i = index$ to $m$ do
28:     remove $Sc.event[i]$
29:   end for
30: the number of events in scenario $Sc = index$
31: end if

Figure 6-11: Decomposing a scenario by creating a new scenario for the overlapping part and removing the overlapping part. The decomposition depends on the position of the overlapping part specified by $index$. It also depends on $count$ which is the number of overlapping events.
Chapter 7

Behavioral Model with Alternative Alphabets

7.1 Introduction

Scenarios are written descriptions used to organize the information generated during requirements analysis [8]. These descriptions are typically in story form giving a sequence of events. Representing user requirements as a set of scenarios is an established activity in software engineering [3, 4] and its use makes the communication between end users and software developers more efficient [9]. To assist with representing scenarios in intuitive and clear forms, sequence diagrams, such as the Unified Modeling Diagrams (UML) [8] and Message Sequence Charts (MSC) [11], are used. For instance, Figure 7-1 [13] shows the MSC corresponding to the interactions between user, cache and server.

In addition to scenarios, properties can be used to add constraints and/or conditions that limit the behavior of a system [14, 15]. Properties are used to ensure that the system will not exhibit incorrect or disallowed behavior. Properties are often represented in forms such as the Object Constraint Language (OCL) [16] and Fluent Linear Temporal Logic (FLTL) [15]. For instance, Figure 7-2 [13] and Figure 7-3

68
Figure 7-1: A scenario, represented as an MSC, that shows the interactions between user, cache and server.

<table>
<thead>
<tr>
<th>Action</th>
<th>Pre-condition</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestCacheData</td>
<td>requestPending = false</td>
<td>requestPending = true</td>
</tr>
<tr>
<td>responseCacheData</td>
<td>requestPending = true and cached = true</td>
<td>requestPending = false</td>
</tr>
<tr>
<td>requestServerData</td>
<td>requestPending = true and cached = false</td>
<td></td>
</tr>
<tr>
<td>responseServerData</td>
<td>post: cached = true</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-2: Constraints represented as an OCL on the scenario shown in Figure 7-1.

A behavioral model is a formalism that describes the behavior of the system using a sequence of state transitions [9]. It also provides a graphical representation for the new system to be developed [9]. Scenarios and properties can be used to create a behavioral model for the system to be developed [21] and such a model can be used to validate the user requirements [21, 37]. Similarly, scenarios may also be used to validate both system analysis and design [38].
In [14], Uchitel et al. introduce an algorithm to synthesize a behavioral model from scenarios and properties. This algorithm constructs the partial behavioral model in three steps. Figure 7-4 illustrates these steps. First, it creates two separate behavioral models one for the scenarios and another for the properties. The resulting behavioral models are represented as Label Transition System (LTS)\(^1\). This step will be referred to, in this work, as creation. Next, the algorithm converts the behavioral models to partial behavioral models represented as Model Transition System (MTS)\(^2\). This step will be referred to, in this work, as conversion. Finally, the algorithm merges the partial behavioral models of both scenarios and properties. This step will be referred to, in this work, as merging.

Algorithms, that synthesize behavioral models from scenarios and properties, such as Uchitel et al.’s approach [14] and Krka et al.’s approach [13], do not consider alternative alphabets [14] if the known set of events before and after the creation of the behavioral models are different. When alternative alphabets occur, these algorithms must be reapplied because they assume that the system’s alphabet will not be affected by the addition of new requirements. However, this assumption is not practical because the requirements specification is usually built incrementally. New scenarios and new properties are expected to appear during the requirements specification process. Both new scenarios and new properties are likely to have new alphabets. A synthesis

\(^1\)Label Transition System (LTS) is a formalism used to describe the behavioral model.
\(^2\)Model Transition System (MTS) is a formalism used to describe the partial behavioral model.
Figure 7-4: The synthesis algorithm generates a partial behavioral model in three steps: creates behavioral model, converts it to partial behavioral model, and merges both the scenarios and properties partial behavioral models.

algorithm supporting alternative alphabets would offer a significant improvement over the mentioned algorithms because it will eliminate the re-run of the creation step.

In the case of alternative alphabets, the known alphabet is vital in the conversion step during the generation of the partial behavioral model [14]. This characteristic is because during the conversion step, the synthesis algorithm assumes the system’s alphabet will not be affected by the addition of a new requirement. Then, it creates transitions based on this assumption for the generation of the transitions for its behavioral model. However, in the case of alternative alphabets, such alphabet will be affected by the addition of a new requirement. This means the generated behavioral models are not valid for modification and they need to be reconstructed.

A re-run of the creation step can be avoided by modifying the generated behavioral models to incorporate an alternative alphabet. Our decision to modify the behavioral models instead of reconstructing them is because the creation step is known to be the most expensive component of the synthesis algorithm [29].

In the following sections, we show how to modify both the partial behavioral model of the scenarios and the partial behavioral model of the properties. The solution proposed in this work is applied to the models generated during the creation step and/or conversion step. Our approach is composed of two fundamental components. One to address the new actions discovered in the new properties and another to address the new actions discovered in the new scenarios. This approach enhances the
capabilities of the existing algorithms for synthesizing partial behavioral models by addressing the alternative alphabets.

This chapter is organized as follows. Section 7.1 introduces the necessary background on properties and scenarios, properties notations, and behavioral models notations. Section 7.2 defines the notation of alternative alphabets and illustrates the modification of behavioral models to reflect newly discovered events. In Section 7.3 and Section 7.4, we show how to modify the partial behavioral model of the properties, and partial behavioral model of scenarios to reflect newly discovered events, respectively. Conclusions are given in Section 7.5.

7.2 Modifying Behavioral Models to Reflect Newly Discovered Events

In this section, we explain the meaning of two terms, alphabet and alternative alphabets. Next, we explain the effect of alternative alphabet in the synthesis algorithm. We also explain the meaning of communicating alphabet and superset alphabet. Finally, we show the different cases which arise when a new alphabet is discovered.

Definition 11 and Definition 12 give the definitions of alphabet and alternative alphabets, respectively.

Definition 11 (Alphabet). An alphabet is a set of actions that is used in both scenarios and properties to describe a system.

Definition 12 (Alternative Alphabets). Let $\alpha_1$ be the alphabet that was used during the creation of a partial behavioral model and let $\alpha_2$ be the alphabet $\alpha_1$ union the new actions discovered when a new scenario or property is added. We call $\alpha_1$ and $\alpha_2$ alternative alphabets.

Alternative alphabets arise when new actions in a new scenario (or a new property)
are discovered and need to be included. Modifying the partial behavioral model, in the case of alternative alphabets, presents a problem\(^3\), and in this case, there are two options for solutions. We could either (1) restart the synthesis algorithm from the beginning, creating a new model or (2) modify the existing model [14]. Alternative (2) avoids the exponential growth results of the computation time of synthesizing partial behavioral model from a larger number of scenarios and properties with a larger alphabet [22].

Each scenario or property exhibits a set of actions which may be described as a *communicating alphabet*. We will consider the set of actions recognized by all scenarios (and properties) and call it the *superset alphabet*. Definitions 13 and 14 conclude the definitions of the communicating alphabet and superset alphabet, respectively.

**Definition 13** (Communicating alphabet). A *communicating alphabet*, for a scenario (or a property), is the set of events that are exhibited by a scenario (or a property).

**Definition 14** (Superset alphabet). A *superset alphabet* is the set of all events that are recognized by all scenarios and properties.

Similar scenarios (or properties) may be described by different alphabets originating from different stakeholders. This characteristic is because individual viewpoints and vocabularies can be expected to differ [14]. Usually a synthesis algorithm is used to build a partial behavioral model from a set of scenarios and a set of properties incrementally. As a consequence, it is common that new scenarios and new properties are added during the requirements engineering and model building stages [14]. Often, these new scenarios and new properties have new actions that were not considered in a previous application of the synthesis algorithm. Therefore, it is important to find a way to incorporate the newly discovered actions into the set of the known actions [14].

---

\(^3\)The problem is that the synthesis algorithm assumes the alphabet is fixed. However, in the case of alternative alphabet such characteristic does not hold.
Any new event is either originated from a scenario or a property. Also, two types of behavioral models are generated for the system requirements. The first type is for the scenarios and the other type is for properties. Therefore, when a new event is added, the designer may be faced with the following semantics.

1. Modifying the partial behavioral model of scenarios to reflect newly discovered events in scenarios. New scenario is added to describe a new behavior.

2. Modifying the partial behavioral model of properties to reflect newly discovered events in scenarios. New scenario is added to describe a new behavior.

3. Modifying the partial behavioral model of scenarios to reflect newly discovered events in properties. New properties are needed to state the restrictions associated with scenarios.

4. Modifying the partial behavioral model of properties to reflect newly discovered events in properties. Additional properties were identified.

Case (2) and case (4) are discussed in Section 7.3. Case (1) and case (3) are discussed in Section 7.4.

7.3 Modifying the Partial Behavioral Model of the Properties to Reflect Newly Discovered Events

The modification that is needed to be applied to the partial behavioral model of properties when the new actions are originated from a new property is the same.
as the modification that is needed when the new actions are originated from a new scenario. Therefore, we will explain both case (2) and case (4) in this section.

The current approaches, such as Uchitel et al.’s [14] and Krka et al.’s [13], that consider converting properties into behavioral models, do not consider two issues: The definition of the alphabet used in properties and the usage of the temporal logic used in FLTL formalism. We start by explaining the definition of alphabet in properties. Next, we compare two types of temporal logic used in FLTL formalism. After that, we explain the proposed modification for the behavioral model of properties. Finally, we provide an example.

7.3.1 The Definition of The Alphabet Used in Properties

The property’s alphabet is the actual set of events used by the property [29] which we call the communicating alphabet (see Definition 13). This means it is possible (and more likely) that a property will have its own (different) communicating alphabet. Therefore, alternative alphabets for the set of properties exist in many cases.

**Proposition 1.** [29] Let $a_1$ and $a_2$ be safety properties in FLTL. If $a_1$ and $a_2$ are closed under stuttering\(^6\) then for all traces $tr \models (a_1 \land a_2)$ if and only if $tr$ is accepted by constraint $a_1 \parallel$ constraint $a_2$.

In [29], the Proposition 1 is introduced to describe the relationships between the composition of properties and the composition of their behavioral models. Proposition 1 implies that the property’s communicating alphabet includes only the events used by that property and cannot be the universe of all actions (the superset alphabet). If one supposes that the property’s communicating alphabet is the universe of all events in all properties (the superset alphabet) then the use of Proposition 1 will

---

\(^6\) *This will be explain in the Section 7.3.2.2*
not be applicable. As a consequence, the creation of behavioral model of multiple properties will not be possible.

7.3.2 Comparing Two Types of Temporal Logic

In this section, we describe two types of temporal logic used in FLTL formalism. We explain the closed under stuttering notation. We show which temporal logic is more appropriate to be used in the FLTL formalism to describe the system’s properties.

7.3.2.1 Types of Temporal Logic

In FLTL formalism, two types of temporal logic can be used to describe the system requirements as properties: Lamport temporal logic [27, 28, 46] and the extended temporal logic [15]. The temporal logic described by Lamport [27, 28, 46] uses the boolean operators: and ∧, or ∨, not ¬, and implication → and the temporal operators: always □, and eventually ◇ [28].

Extended temporal logic is defined using the standard boolean operators: and ∧, or ∨, not ¬, and implication → and the temporal operators: next X, strong until U, weak until W, eventually ◇, and always □ [15].

The main difference between Lamport temporal logic and extended temporal logic is the usage of the X operator. The effect of the usage of this operator will be discussed later in Subsection 7.3.2.3.

7.3.2.2 Closed (invariant) Under Stuttering Definition

Suppose, for instance, that we have the following two sequences:

σ₁ : S₀ \xrightarrow{α₁} S₁ \xrightarrow{α₂} S₂ \xrightarrow{α₃} ... and

σ₂ : S₀ \xrightarrow{α₁} S₁¹ \xrightarrow{α₂¹} S₁² \xrightarrow{α₂²} S₁³ \xrightarrow{α₂³} ... \xrightarrow{α₂ⁿ} S₂ \xrightarrow{α₃} ...

where $S_i$ represents a state, and $α_i$ is an event that belongs to the property commu-
nivating alphabet. If the temporal logic can not distinguish between $\sigma_1$ and $\sigma_2$, then the temporal logic is closed (invariant) under stuttering. Otherwise, it is not closed (variant) under stuttering [27]. In the other words, if $\sigma_1$ and $\sigma_2$ always have the same truth values, then the temporal logic is closed (invariant) under stuttering. On the other hand, if $\sigma_1$ and $\sigma_2$ may have different truth values, then the temporal logic is not closed (variant) under stuttering [27].

7.3.2.3 What Types of Temporal Logic are More Appropriate to Use to Describe Properties (The Effect of Using the Next Operator)

The temporal logic operator $\text{next}$, $X$, is used to express that an event occurs in the next state. However, using the $\text{next}$ operator, $X$, in the event-based temporal logic makes it not closed (variant) under stuttering [27].

The extended temporal logic uses the $\text{next}$ operator, $X$, which makes it not closed (variant) under stuttering. As a consequence, the creation of the compositional behavioral model for a set of properties is not possible. The reason for this is because Proposition 1 [27, 29] is not applicable. However, under some assumptions the above condition, closed under stuttering, can be satisfied such as in [29]8. On the other hand, Lamport temporal logic does not use the $\text{next}$ operator, $X$, which makes it closed (invariant) under stuttering. As a consequence, the creation of the compositional behavioral model for a set of properties is possible because Proposition 1 is applicable [27, 29].

The use of the $\text{next}$ operator $X$, makes the temporal logic more expressive, but this makes it not closed (variant) under stuttering. The ability provided by the addition of the $\text{next}$ operator $X$ can be substituted by the use of other operators [27].

---

7 The temporal logic that we mean here is synchronous temporal logic and not asynchronous temporal logic.

8 This is done by assuming that asynchronous temporal logic is used. However, in event-based model synchronous temporal logic should be used.
This means the elimination of the \textit{next} operator, $X$, is possible without much loss of expressiveness or preciseness of the temporal logic [27].

We propose the use of Lamport temporal logic [27,28,46]. Our choice is because this logic is closed under stuttering and therefore, allows us to use the parallel composition of the behavioral model of the individual properties to create the behavioral model of all properties.

\subsection*{7.3.3 The Proposed Algorithms}

The creation of the behavioral model of the properties, represented as an LTS model, is affected by its communicating alphabet and the difference between the superset alphabet and its communicating alphabet. If we add an extra event, say, for instance, other-events\footnote{This is similar to the term \textit{anon} used in [15] where is used to represent other events (actions) that the system may use but is not used in defining properties.}, to the superset alphabet, then we can use it as a reference for any new alphabet discovered during the elaboration of the system requirements. Therefore, any addition of a new alphabet will not affect the created behavioral model of properties. Exception to this is when we refer to the difference between the superset alphabet and the property’s communicating alphabet. In this case the extra event, other-events, can be used as a reference to update the behavioral model.

Our idea is to keep the behavioral model of properties. Then, we make the modification that we propose in Figure 7-5. This can be applied for each individual behavioral model of a property. Such modification can also be done for the conjunction of behavioral models of properties represented as an LTS model. The reason for this is because building a behavioral model for the conjunction of properties is equivalent to the conjunction of the behavioral model of properties as Proposition 1 states [14]. Algorithm [14], shown in Figure 5-7, creates a partial behavioral model for properties without considering alternative alphabets. To modify this algorithm to
take into consideration the alternative alphabets, we propose the following: (1) redefi-
ne the alphabet to be the alphabet union a new term, other-events. (2) in the case
of alternative alphabets, modify the behavioral model that resulted from the creation
step (instead of re-applying the creation step) (see Figure 7-4) by applying the step
shown in Figure 7-5 on the existing partial behavioral model of properties. This can
also be done on the partial behavioral model that resulted from the conversion step.

1: Let new-actions be the new actions discovered in new properties or new scenarios
2: for all state $S_1$ such that $S_1 \xrightarrow{other-events} S_2$ do
3:    Add new transition $S_1 \xrightarrow{newactions} S_2$
4: end for

Figure 7-5: Modify the partial behavioral model of properties to incorporate
the new actions that are discovered.

7.3.4 Example

Suppose, for instance, a television system which blanks its screen while it tunes
to a new channel [15]. The properties for such a system are described in Figure 7-7.

In the following, we show the effect of adding a new alphabet to an already created
behavioral model of properties. We modify the alphabet for the system in Figure 7-7
and examine the resulting effects both with and without the use of the solution this
paper proposes.

7.3.4.1 The Effect of Adding a New Alphabet on the Behavioral Model
of the Properties Without Using the Proposed Algorithm

The alphabet of the properties in Figure 7-7 is $\alpha = \{\text{blank, unblank, endtune,}
tune\}$. Figure 7-8(a) shows the behavioral model of the above properties. Now
suppose, for instance, that new events (actions), Press-sound-up-key, Press-sound-
Figure 7-6: The modified synthesis algorithm. This algorithm generates a partial behavioral model, for both scenarios and properties if the new actions are discovered in properties.

\[
\text{fluent BLANKED} = \langle \text{blank, unblank} \rangle \\
\text{fluent TUNING} = \langle \text{tune, endtune} \rangle \\
\text{NOARTIFACTS} = \Box(\text{TUNING} \Rightarrow \text{BLANKED})
\]

Figure 7-7: Properties that describe a television system which blanks its screen while it tunes to a new channel [15].

down-key, are discovered in a new scenario or property. The new alphabet for this case is \(\alpha_2 = \alpha_1 \cup \{\text{Press-sound-up-key, Press-sound-down-key}\}\) or \(\alpha_2 = \{\text{blank, unblank, endtune, tune, Press-sound-up-key, Press-sound-down-key}\}\). As it can be seen the result of this operation created the alternative alphabet \(\alpha_2\). As a consequence, the behavioral model must be re-created from scratch according to the new alphabet \(\alpha_2\) and the conversion algorithm needs to be reapplied also according this new alphabet. The new behavioral model obtained by using the new alphabet, \(\alpha_2\), is shown in Figure 7-8(b).

7.3.4.2 The Effect of Adding a New Alphabet on the Behavioral Model of the Properties Using the Proposed Algorithm

If we use the proposed algorithm, then \(\alpha_1 = \{\text{blank, unblank, endtune, tune, other-events}\}\). The behavioral model of this property is shown in Figure 7-9(a). Now suppose, for instance, that new events (actions), \(\text{Press-sound-up-key, Press-sound-down-key}\), are discovered in a new scenario or property. The new alphabet for this case is \(\alpha_2 = \alpha_1 \cup \text{newEvent}\) or \(\alpha_2 = \{\text{blank, unblank, endtune, tune, other-events, Press-}\)
(a) Before the inclusion of the new alphabet.

(b) After the inclusion of the new alphabet.

Figure 7-8: The behavioral model of the set of properties without using the proposed algorithm.

(a) Before the inclusion of the new alphabet.

(b) After the inclusion of the new alphabet.

Figure 7-9: Using the proposed algorithm, Figure (a) shows the behavioral model of the set of properties before the inclusion of the new alphabet, and Figure (b) shows the behavioral model of the set of properties after the inclusion of the new alphabet.

sound-up-key, Press-sound-down-key}. The advantage of the proposed algorithm is that we do not need to re-create the behavioral model. Instead, we can add the new events to each transition that has other-events as a label. This process will save the need of re-applying of the costly step for the creation of the behavioral model of the property. The modified behavioral model according to the new alphabet, \( \alpha_2 \), is shown in Figure 7-9(b).

If the event is originated from a scenario, then no other modifications are needed to the behavioral model of the properties. This is because the behavioral model of properties will not be affected of a new scenario. On the other hand, if the event is originated from a property, we need an extra step to generate a behavioral model for the new property and merge it with the existing model. This is because the behavioral model of properties need to include the new behavior introduced by the new property.

81
7.4 Modifying the Partial Behavioral Model of the Scenarios to Reflect Newly Discovered Events

The modification of the behavioral model when new events are originated from scenarios, case(1), is different than the modification when new events are originated from properties, case(3). Therefore, we will discuss each case separately. First, in Subsection 7.4.1, we discuss case(1). Then, in Subsection 7.4.2, we discuss case(3).

7.4.1 Modifying the Partial Behavioral Model of the Scenarios to Reflect Newly Discovered Events in Properties

To modify the partial behavioral model of scenarios to take into account the effect of new actions that are discovered in new properties, we propose the algorithm shown in Figure 7-10. In the algorithm shown in Figure 7-10, sink is a special state in the partial behavioral model of scenarios, and $Y$ is an event that belongs to the superset alphabet. This means $Y \in$ Superset Alphabet. Suppose, for instance, that we have the partial behavioral model shown in Figure 7-11(a). Also, suppose that $e$ is a new action discovered in a new property. Then, according to the algorithm shown in Figure 7-10, we modify the partial behavioral model shown in Figure 7-11(a). This

1: Let new-actions be the new actions discovered in the new properties
2: for all state $s_1$ such that there is a transition $s_1 \xrightarrow{Y} \text{sink}$ do
3: Add new transition $s_1 \xrightarrow{\text{new-actions?}} \text{sink}$
4: end for

Figure 7-10: Modify the partial behavioral model of scenarios to incorporate the new actions are discovered in properties.

10See Section 7.2 for both case1 and case 2
modification introduces the transitions: $S_1 \overset{e?}{\rightarrow} S_0$, $S_2 \overset{e?}{\rightarrow} S_0$, and $S_3 \overset{e?}{\rightarrow} S_0$, as shown in Figure 7-11(b).

Figure 7-11: A partial behavioral model for a set of scenarios before addition of new alphabet, and a partial behavioral model for a set of scenarios after addition of new alphabet “e”, Figures (a), and (b) illustrate the first and second model respectively.

### 7.4.2 Modifying the Partial Behavioral Model of the Scenarios to Reflect Newly Discovered Events in Scenarios

Although it is possible that a new scenario is added after the creation of the partial behavioral model, the existing algorithms such as ([17, 18, 21]) do not support adding the effect of a new scenario (even if it has the same alphabet) to the partial behavioral models. This situation arises because the creation of the behavioral model depends on

Figure 7-12: The modified synthesis algorithm. This algorithm generates a partial behavioral model, for both scenarios and properties if the new actions are discovered in scenarios.

the way the scenarios are related to each other\textsuperscript{11}. When adding a new scenario, it is

\textsuperscript{11}Each scenario contains interactions for all entities in the scenario. On the other hand, the behavioral model is created for each entity.
difficult to determine how the new scenario is related to other scenarios. Even if it is known how the new scenario is related to the other scenarios, it is difficult to modify the partial behavioral model to reflect the effect of the new scenario. Therefore, a possible way to add a new scenario is to re-apply the steps from one to three (see Figure 7-4). Figures 7-6 and 7-12 illustrate the modified synthesis algorithm that generates a partial behavioral model if the new events are discovered in properties or scenarios, respectively.

7.5 Conclusion

The behavioral model of properties can be modified using the proposed algorithm instead of re-constructing it. The behavioral model of scenarios can be modified using the proposed algorithm if the actions are originated from properties instead of re-constructing it.

Alternative alphabets cause the algorithms, that assume fixed alphabet to re-run again by re-generating the behavioral model of both scenarios and properties. Our proposed algorithm eliminates the need of the re-run of the creation step of the behavioral models of both scenarios and properties.

Instead, our proposed algorithm modify the already created behavioral models. The elimination of the re-run of the creation step is important because this step is known to be a costly step.
Chapter 8

Summary and Conclusions

8.1 Summary

Scenarios and properties are used to represent requirements specification. Scenarios usually have overlapping parts (two or more scenarios have shared parts). Behavioral models are synthesized from scenarios and properties. In the case of overlapping scenarios, behavioral models can be sub-optimal (having extra states and transitions). We proposed algorithms to detect the overlapping parts, then decompose scenarios that contains the overlapping parts. We used pattern matching technique to detect the overlapping scenarios.

The system alphabet is the set of actions or events used to describe scenarios and properties. Alternative alphabets result from the difference between the alphabets, before and after the generation of the behavioral model when adding a new property or a new scenario to the requirements set. We propose algorithms to reflect the effect of the new alphabet on the behavioral models instead of re-run the algorithms from the start to include the new alphabet. Incorporating new alphabet to an existing behavioral model of property was achieved by adding a new term to the system alphabet. Then developing an algorithm that uses this term as a pointer to add the new alphabet discovered. Incorporating new alphabet to an existing behavioral model
of scenarios was achieved by developing an algorithms that adds the new alphabet to transitions that go to a special state in the behavioral model, the sink state.

8.2 Validation

The main goal of this dissertation are:

- decrease the overlaps between scenarios which will result in a more optimal behavioral model, and

- modify the behavioral models of both scenarios and properties to include new alphabets instead of reconstructing these models.

In the following we repeat the claims that we have made in Chapter 1. Then, we explain how these claims have been accomplished through the dissertation.

1. **overlapping scenarios can be detected using pattern matching.**

   As we have illustrated in Section 6.3 shared events/actions among scenarios were detected using a pattern matching algorithm. The algorithm records all shared events/actions in a table called overlapping events table. Table 6.8 in Section 6.3 illustrates overlapping events table. The correctness of pattern matching algorithms have been proven.

2. **decomposing scenarios can reduce the overlaps between scenarios.**

   In Section 6.4, we provided algorithms and demonstrated examples which show how decomposing scenarios reduce the overlapping between scenarios. See, for instance, Tables 6.11, 6.12, 6.13 and 6.14.

3. **detection and decomposition of the overlapping scenarios lead to more streamlined behavioral models.**
Generating a behavioral model from scenarios without overlaps will not produce extra transitions and states. This is because the source of extra transitions and states is overlapping scenarios. Reducing overlapping will reduce the extra transitions and states.

4. modifying behavioral models instead of recreating them, when alternative alphabets occur, can be achieved.

We showed in Chapter 7, how the inclusion of new alphabets in the behavioral models was achieved without reconstruction these models. Figures 7-9 and 7-11 illustrate two examples that perform the inclusion of the new alphabet in the behavioral models without the need of reconstructing them.

Our research consists of developing algorithms and validating the results by using a set of studies. Therefore, our research approach is consistent with that described by the survey of software engineering research in Shaw [47].

8.3 Conclusions

The elimination of overlapping scenarios was approached by the use of pattern matching. The algorithms proposed structured the solution in two stages, the identification of the overlapping and the decomposition of the scenarios. Although the number of scenarios may increase, the overall number of events is minimized. The strategy proposed made possible the creation of behavioral models that are free from the redundancies caused by the presence of the overlapping scenarios. The behavioral model of properties is modified using the proposed algorithm instead of re-constructing it. The behavioral model of scenarios is modified using the proposed algorithm if the actions are originated from properties instead of re-constructing it. Alternative alphabets cause the algorithms, that assume fixed alphabet to re-run
again by re-generating the behavioral model of both scenarios and properties. Our proposed algorithm eliminates the need for the re-run of the creation step of the behavioral models of both scenarios and properties. Instead, our proposed algorithm modify the already created behavioral models. The elimination of the re-run of the creation step is important because this step is known to be a costly step.

8.4 Future Work

Finding similar patterns can also be achieved using sequence comparisons. Using sequence comparisons to detect overlapping scenarios and compare it with pattern matching can be an extension for this work. Explore the opportunities to modify the behavioral model of scenarios when alphabet discovered in scenarios is another possible extension to this work. Developing a tool to employ the proposed approach is another possible extension.
References


