COMPOSITION AND VERIFICATION OF FORMAL BEHAVIORS

Siamak Kolahi

A thesis in the Department of

Computer Science

and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

June 2008

Your file Votre référence
ISBN: 978-0-494-42532-9
Our file Notre référence
ISBN: 978-0-494-42532-9

Abstract

Composition and Verification of Formal Behaviors

Siamak Kolahi

In this thesis, we present a platform and a tool support for formal modeling, automated composition, and formal verification of partial system behaviors defined as Use Case Automata (UCAs). Based on research works [24, 31, 32], the Use case Composition, Modeling and Verification (UCOMV) is presented as tool support for the visual modeling of formal behaviors and their merging through the notion of composition expression. The composition expressions determine the extension points in the use cases where the composition is performed, and the operators for the semantics of the composition. The theory and the tool supports a new incremental approach of building the desired system specification with a formal automated mechanism of composition. In addition, it allows the verification of UCAs over temporal properties defined as part of the system requirements specification using an integrated model verification tool. The UCOMV also provides features on traceability of the requirements, and an XML based schema for the introduction of new operators for UCAs composition.

Acknowledgments

I would like to express my sincere regards and appreciations to Professor Rachida Dssouli and Professor Aziz Salah for all their supports and assistances during this long path. Also I would like to thank Mrs. Rabeb Mizouni for all her kind corporations, and her friendship. And the foremost, to my dearest sister, Solmaz Kolahi for all her supports, helps and caring, indeed.

I dedicate and owe all this, to my beloved parents...

Contents

List of Figures x

1 Introduction 1

1.1 Motivation 1

1.2 Problem Definition 2

1.3 Solution 3

1.3.1 Context 4

1.3.2 Objective 4

1.4 Thesis Plan 4

2 Background 6

2.1 Introduction 6

2.2 Basic Concepts 7

2.2.1 Requirement Engineering 7

2.2.2 Use Cases and Scenarios 9

2.3 Inherent Issues 11

2.4 Conclusion 17

3 State of the Art **18**

3.1 Introduction 18

3.2 Scenario Notations 19

3.3 Scenario Composition 21

3.4 Supporting Tools 24

4 Use Case Automata **29**

4.1 Introduction 29

4.2 The UCA Model 30

4.3 Approach Overview 34

4.4 Composition 37

4.5 Operators 42

4.6 Conclusion 48

5 UCOMV **50**

5.1 Introduction 50

5.2 Integrated Framework 51

5.3 Tool Architecture 52

5.4 Three-Layered Design 53

5.5 Conclusion 55

6 Specification Interface **56**

6.1 Introduction 56

6.2 Interfacing the Specification 57

6.3 Automata Design Environment 59
6.4 Composition Expression Design 62
6.5 Verification Environment 66
6.6 Variable Definitions 68
6.7 State Coloring 72
6.8 XML Schemas 74
6.9 Conclusion 76

7 Composition Engine **77**

7.1 Introduction 77
7.2 Expression Evaluation 78
7.3 Builder Synthesis 79
7.4 Merging Algorithm 82
7.5 UCA Post-processing 84
7.5.1 Final States 84
7.5.2 Removal Algorithm 85
7.5.3 UCA Minimization 86
7.6 Composition of Extended UCAs 87
7.7 Conclusion 90

8 Model Verification **91**

8.1 Introduction 91
8.2 Behavior Verification 92

8.3 Promela and SPIN 94
8.4 Model Transformation 95
8.4.1 UCA as Promela Model 100
8.5 Verification and Error Trace Parsing 102
8.6 Property Inheritance 106
8.7 Conclusion 106

9 Traceability **108**
9.1 Introduction 108
9.2 UCA Relation Types 109
9.3 Hierarchy of Dependency 110
9.4 Forward Propagation 111
9.5 Optimal Propagation 113
9.5.1 Compositional Order 114
9.5.2 Forward Propagation Algorithm 115
9.6 Forward Retrieve of Extension Points 116
9.7 Conclusion 118

10 Case Study: e-Purchasing System **119**
10.1 Introduction 119
10.2 Partial Requirement Specification 120
10.3 Composition Increments 122
10.4 Behavioral Verification 125

10.5 Conclusion 128

11 Discussion, Conclusion and Future Works **129**

A Intermediate UCAs **134**

Bibliography **137**

List of Figures

1 Basic MSC and its corresponding implementation 13

2 Example UCAs 31

3 Clones of the two example UCAs 34

4 Composition Approach[31] 35

5 Expected UCA from evaluation of an expression 37

6 Builder example for UCA X 40

7 State and transition extension points and their affected traces 44

8 Concept of Include builder (a), the example UCA X (b), and the generated builders before and after transition b (c),(d), and on state (e) . 46

9 Overview of the operators composition semantics [32] 47

10 Tool Component Architecture 54

11 UCA-EUCA management environment 58

12 Automata Design Multi-Page Editor 60

13 Overview of UCA and EUCA data structures 61

14 Derivation of composition expression classes 64

15 Class hierarchy of extension points 65

16 Defining transition based composition expression 66
17 Model checking environment 67
18 LTL Property Manager 68
19 Class structure of the variable operations 70
20 Class Structure of variable comparators 70
21 Adding conditions and assignments for an extended transition 71
22 Global and local variable list 72
23 UCOMV 73
24 Derivation of Operator classes 81
25 Derivation of transition remover classes 90
26 Example of an UCA, and its Promela models in UCOMV 99
27 Hierarchical Dependency Chart 111
28 Primitive UCA specification of e-Purchasing system 121
29 Product Selection 2 after three increments 127
30 Prod-select-1 and Ordering-1 UCAs 135
31 Ordering-2 UCA 135
32 Ordering-3 UCA 136

Chapter 1

Introduction

1.1 Motivation

Modeling and reasoning about a prospective system is a key solution to todays complex system development problems. Producing a sound, complete and consistent specification for the system is an important step which can help studying the system a-priori and largely affect the future steps of the development process. It can help to detect and reduce defects in early stages of the system life-cycle, and hence, reduce the potentially large cost of detecting these defects in future steps of system development. In fact, it is the place where the real complexity of the system begins to form.

Scenario based behavior analysis is one of the most accepted and widely studied approaches to produce the desired system specification. Scenarios are partial narrators of the system behavior story, which in case all together can show how the system

works. Each scenario describes a specific system behavior from users perspective, and can show interactions between system and its users. It can also show how different parts of the system interact in order to provide the required functionality. They are abstract and understandable for the stakeholders, while expressive enough to provide the ground for representing large and complex system specifications. Moreover, a set of these scenarios form a use case which act as a system-level functionality. Each use case depicts a desired functional requirement of the prospective system, carrying a set of execution scenarios of the behavior.

1.2 Problem Definition

Despite consensus on their usefulness, use case and scenario based approaches suffer from the lack of a rigorous and easy-to-use formality. The formality could trigger the development of automated composition and verification mechanisms that can support generation of proper system specifications. The complexity of such approaches lies in finding a formal model with proper level of formality for the presentation of behaviors, defining an elaborative composition mechanism, preventing implied and undesired behaviors stemming from the composition, and verifying the model for safety and progress properties.

Moving to more a formal presentation model for the scenarios and use cases would increase the expressiveness power and facilitate automating the composition and verification of system models. However, as the model gets more formally expressive, it gets harder to understand for the stakeholders and more complex to compose. Moreover, regardless of the notations used, the composition of use cases is a hard task and is subjected to many inherent issues that the designer should deal with. Issues such as realizability of the model, state explosion problem, and implied scenarios generated from the integration are examples of such unresolved issues.

1.3 Solution

Modeling the system as a whole is a hard and complex task to do. Moreover is harder to define proper verification properties on the model. Deploying an incremental approach of constructing the system specification can help to reduce the problems on this suit. It can break the complexity of modeling and verification of the system to small problems in each increment in the specification. Such an approach could assure to obtain the verified system model and gradually form the desired trustable specification. Our objective is to facilitate this process. We present a formal description of use cases, provide mechanism for automated composition of them, and develop a platform for defining and verifying behavioral properties on the generated models. We also present a supporting tool for the deployment of such approach over large scale real-life specification problems.

1.3.1 Context

The originality of the approach presented in this thesis is based on a new incremental approach for modeling, composing and verifying the behavioral models. The theory of the modeling and composition refers to the works of the PhD thesis of Rabeb Mizouni [31] and our earlier research work [32, 33].

1.3.2 Objective

In this thesis, we present the theory of use case composition. Furthermore, we develop a supporting tool as a proof of concept of the theory. We also use temporal logic for the verification of use cases. Moreover, using the supporting tool, we apply the theory on a real-life specification problem. Novel features such as traceability have also been presented.

1.4 Thesis Plan

The thesis is organized as follows. In Chapter 2, we discuss the basic concepts and problem definition for the scenario based requirement analysis. We also present some of the most well-known inherent issues in the field. In Chapter 3, we study the state of the art in the literature for scenario presentation and composition, and see some of the relevant works on tackling the problems in the area. We also review some of the available tools for supporting novel theoretical works. Chapter 4 presents UCAs as our formal model of system behaviors, and relevant theoretical basis are given

for the composition of UCAs. We also discuss the EUCA as the extended model of UCA, and novel concepts on introducing new compositional semantics desired compositions. Chapter 5 discusses the UCOMV as the tool support for the theory and its architecture. Different layers of the tool architecture discussed in this chapter are detailed in its following chapters. Chapter 6 discusses the design environment for the specification with useful features provided in the tool for the modeling and verification of UCAs in the tool. Furthermore in Chapter 7 we detail the design of the composition engine of the tool, which is the place that our core composition theory has been implemented. The model verification and its integrated environment in the tool is discussed in Chapter 8. This chapter also discusses the model transformation of UCA to the target language and the limitations in this process. Chapter 9 address an interesting and essentially useful feature in the tool which supports the traceability of composed behaviors in the generated model. It describe how the changes in the specification are propagated in the hierarchy of their composition. We show the applicability of our approach in Chapter 10 on the generation of the specification for an e-purchasing system, and give examples of possible behavioral properties which could be verified on the generated models for such system. Chapter 11 discusses the benefits of our approach as well as its limitations in facing real-life specification problems. We conclude our discussion and show the possible directions for future extensions of our research motivation in this chapter.

Chapter 2

Background

2.1 Introduction

Scenario based requirement analysis is one the most accepted and welcomed approaches to produce the desired system specification. Scenarios are partial executions of system models which shows how the system would works with its surrounding environment. They can also show how different parts of the system interact to provide the required functionality. A set of system scenarios combined together can perform required behavior and submit intended results for the user. They conform complete system specification. However integrating and reasoning about scenario sets is hard. Integration is subjected to problematic issues such as realizability and implied scenarios which should be decided to deal with. Moreover, the verification carries the inherent problems on exponential growth in the state space of the system. In this chapter, we present the basic concepts on the area of scenario based requirement

analysis, and address some of the major issues which arise during the modeling and reasoning in such approaches.

The chapter is organized as follows. In Section 2.2 some of the basic concepts for the formal analysis of requirements are provides. In Section 2.3, some of the major issues in the domain of formal system analysis are addressed, and some relevant approaches for tackling these issues are studied. Some of these issues impose major major constraints on a rigorous study of system behaviors.

2.2 Basic Concepts

2.2.1 Requirement Engineering

Requirement Engineering is the science of using technology centered systems for elicitation of the expectations of the stakeholders form a prospective system [8]. During the analysis phase of system development, which in our case is software development, requirements of the system should be clarified. These requirements should cover intended expectations of the stakeholders from the system, and nothing more. As the first step, this phase, plays an important rule and can affects the future steps of the system development. Requirement engineering tries to help this process by using formal and mathematical notations. It includes formalizing the requirements, producing specification documents, developing algorithms for detecting inconsistencies and unintended features in them and in general, building frameworks for requirement

elicitation process. It also provides models from the system which can be used in the future steps of the system development in terms of generative developments and prototypes.

Requirements of a system can be *functional* or *non-functional.* Functional requirements are those regarding the expected behaviors and functions of the system while the latter one concerns how well the system can meet these requirements, e.g. performance,maintainability, usability and to name some of them. In fact, functional requirements show what the system should do and non-functional requirements show minimum accepted standards for the system to do these jobs. These requirements should further be checked for: Validity (are they the right functions regarding the customers needs), consistency (do these requirements conflict at some points and why), completeness (they cover all the functions needed by the customer), feasibility (can they be implemented concerning restrictions on budget, time, and technology?)

Scenario-based requirement engineering is one of the recent and popular approaches in this field. Scenarios describing system views, uses, and services are becoming a common method of capturing functional requirements of reactive and distributed systems. They are particularly appropriate to present behaviors of the system in an understandable way for stakeholders. In this approach, from the early stages of the development, functionalities and behaviors expected from the stakeholders are

defined. They start first by the description of the system as seen by the user in practice. Initially, these functionalities are expressed in a very abstract notation without description on how these functionalities are going to be met. After refining them in an interactive process, we need to move forward to the implementation of the system. Hence more detailed descriptions of the system are needed. Then abstract notations are refined to more detailed ones and sometime more formal ones, and get closer to the details needed for the design phase.

Formal methods are techniques for expressing requirements in a manner that enables the requirements to be studied mathematically. They allow sets of requirements/descriptions to be examined for consistency, and equivalency with respect to requirements/descriptions and compatibility to the original requirements. Formal methods are used to produce formal specifications. A *specification* is a document that clearly and accurately describes requirements of a prospective system. A specification is qualified as formal when it is written using a formal language [8].

2.2.2 Use Cases and Scenarios

Use cases are means for capturing potential requirements of a new system or software. Each use case provides one or more scenarios that convey how a system should interact with end user or other systems to achieve a certain functionality. In the use case presentation, formal expressions are avoided, instead the language used by the end user or domain expert is used to represent requirements. Use cases are sometimes

co-authored by software developers and end users, depending on the level of formality and abstraction of the analysis [34].

A *scenario* is a typical interaction between the user and the system or between two software components. A scenario describes a specific instance of a particular process within the systems business structure. It shows a behavior of the system that may result in an observable outcome for the foreign user or environment. Each scenario sees the system from its own perspective and model the specific behavior of the system in that domain [7]. In this thesis, scenarios and use cases are not two separate concepts. In the object oriented world, use cases are interpreted as classes of related scenarios which perform sequences of actions and yield observable results while scenarios are sequential specific realization of a use case. In this text, we consider scenarios as *specific instances of use cases.*

The composition of these use cases would conform the system behavioral model which is subjected to further checking of model compliance to certain properties and correctness. Despite consensus on their usefulness, use case based approaches suffer from the lack of a rigorous formality. This formality triggers the development of automated composition approaches and verification mechanisms that support the generation of sound and complete system specifications.

2.3 Inherent Issues

Formality Level and Automation

Presenting formal notations of requirements could help the designer to better analyze the intended system and can be further used for model verification, making design revisions, and even code generation. However it needs the analysts and designers to be familiar with mathematical formalizations which are hard to read and understand by the stakeholders. The formality level of our model and the level of the automation of the specification generation is an important influential issue which should be considered and decided. A more formally expressive model would benefit from a more expressive power for the requirements, an automated composition and formal verification mechanisms. A full automated approach can benefit from the ability for rapid prototyping and system generation. On the other hand, a less formally expressive model and semi-automated composition mechanism can benefit from an interactive process which helps refining system specification in case of the detection of implied or missing scenarios with the participation of the stakeholders in an interactive way. It also makes the composition process less complex.

Implementation and Behavioral Model

In a scenario specification, not only a set of acceptable system executions are specified, but also components that participate in these executions and their responsibilities are described, depending on the level of abstraction used in the specification. Thus we

may want to build a set of components which can send and receive messages according to the scenario specifications. Therefore, a model which exhaustively shows the behaviors described by the specification should be developed. The requirements expressed in the scenario specification would be implemented into a specification which contains an exhaustive model of the scenario executions mentioned in the specification.

The synthesized implementation of the system can be used as a precise specification of the intended behavior of the system, as a prototype for exploring the systems requirements and for automated verification of model compliance to the correctness and completeness properties for the system through model checking, or finding possible missing or implied behaviors emerging from the composition. This model can also be used for further refinements through achieving a possible lower implementation of the system.

This implementation and modeling of scenario specifications is done on state based models like *Finite State Machines* or *Label Transition Systems.* Therefore a possible execution of the system, or equivalently the externally observable behavior of a scenario-based specification which are given by a sequence of message labels, are implemented as a trace of the finite state machine.

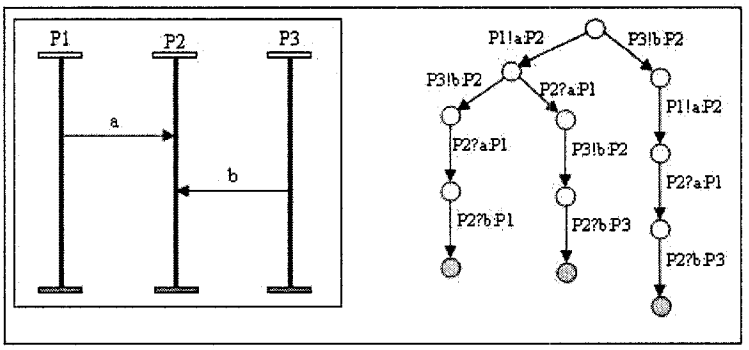

Figure 1: Basic MSC and its corresponding implementation

Realizability

We discussed the benefits of implementing the scenario-based specification into a state-based notation. But this task is not straight-forward and not always possible. A set of scenarios are called *realizable* or *implementable* if a state-base specification can be produced that implement precisely the same behaviors as of the set of scenarios in the specification. It has been shown that in general, the problem of realizability of scenario specifications is an undecidable problem [3] [26], and further only decidable in the NP-Space time complexity [5] [26] [27].

The realizability problem depends on many factors, including the architecture we are using for our implementation model and the semantic type used for the communication model (synchronous vs. asynchronous). There exist two notions of weak and safe realizability [2] depending on whether or not we require the implementation to be deadlock-free. It is shown [3] that for a finite set of scenarios, like MSCs, weak realizability is coNP-Complete while safe realizability has a polynomial time solution.

Implied Scenarios

The composed model from a set of scenarios may, or may not implement exact behaviors noted in the original specification. This is due to the fact that some scenarios may emerge from the composition itself, which may not be expressed in the scenario set. Therefore, after building the implemented model, we need to check the compliance of the model with the original specification. We require a model to include all possible behaviors that have been described in the specification. Not all the behaviors that the implementation is presenting are mentioned in the original scenario specification. In other words, some unintended behaviors may appear in the implementation as the result of composition. We may want to find the closest implementation to our specification with the minimum unintended behaviors. One may expect the model to contain exactly the same behaviors as those specified in the specification, and nothing more. But this implementation does not always exist. One may also not find the implementation that has exactly the same language as the specification. During the composition process, certain unexpected and unspecified behaviors may appear in all possible implementations of the system. These behaviors are called *implied scenarios.*

These behaviors stem from the fact that each scenario in the system has its local view of what is happening in the system, and if sufficient information is not provided, the process may behave unexpectedly and incorrectly in terms of implied behavior at a system level. In fact, implied behaviors are results of an inconsistency between system decomposition and system behavior. Implied scenarios are not produced by

a particular scenario language, say MSC. They are the result of specifying the global behaviors of a system that will be implemented from the processes point of view. They are produced as a result of composing the partial ordered scenarios which has their own view of the system.

Implied scenarios are sometimes unintended. They can simply be acceptable behaviors which have been overlooked by the stakeholders. In this case the scenario specification needs to be revised and completed. In other case however, they represent an unacceptable behavior in which requires the specification be modified to avoid undesired behaviors. Some approaches have been developed to detect implied scenarios [2] [43] [46], and further use them to form the revised specification and hence, more sound and complete specification.

State Explosion

We implement scenarios into a state-based notation in order to have a behavioral model, have explicit formal definition of behaviors, and to verify the behaviors over intended properties and find unintended ones. One big disadvantage for this process is that we have to define all the possible states which the system could reach. This task might be feasible for small systems with limited number of states. However for larger systems, modeling task gets harder and more complex, as the number of intended states grow exponentially resulting in far too more states and transitions to be applicable. Checking the behaviors of the scenario model is usually linear to the size of the states, but the size of the state specification grows exponentially, as the size

of scenario model grows linearly [47]. The base of the exponentiation depends on the number of local states a process has, or the number of values a variable may store. As we have seen, in general, the problem of implementing the state specification is fundamentally hard (NP, PSPACE or worse) and the problem will be how to make this problem scalable [20].

Different approaches have been proposed to tackle the state explosion problem [47]. They include building local state specifications for different parts of the system and then combine them together to have the comprehensive model. But this raises a problem that it is difficult to show the behavior of the whole system from a number of implementations from individual classes. Another widely studied technique is to express symbolically the states of the system and hence, avoid explicit exploration of the exhaustive system model. Other approaches are algorithmic approaches which include state pruning technique to avoid exploring parts of the states space, and techniques to tackle the effects of state explosion such as bit-state hashing, state machine compression catching, etc [20].

2.4 Conclusion

In this chapter, we studied some basic concepts and notations on the area of scenario based requirement analysis which could provide a base for the work in this thesis. Also we have addressed some of the major issues being present in this area and all the approaches on formal modeling and reasoning. In the next chapter, we will briefly overview some of the major research works related to our approach, as well as some tool supports for their practical uses.

Chapter 3

State of the Art

3.1 Introduction

Scenario and use cases are well-known and well-studied concepts in both academia and industry. Many approaches have been investigated to model and reason about scenario specifications. Some of this existing work focus on developing notations for the presentation of scenarios and use cases while others try to tackle problematic issues stemming from the composition and verification. UML Statecharts and MSCs, have become standard notations and are used widely in the industry. They are mostly general purpose notations, which can be applied in many different problems. Some others however, are more specialized notations which could be used in certain domains of system developments.

In this chapter, we present a state of the art literature review on different notations for the presentation and composition of scenarios, as well as tools supporting them.

The chapter is organized as follows: in Section 3.2, we study some of the most well-known notations present in the literature for scenario modeling. In Section 3.3, some of the composition notations and approaches are described and some approaches for determining automated composition are studied. Section 3.4 discuss some of the supporting tools which are available for practicing theories on composition and verification for specification problems.

3.2 Scenario Notations

Many notations [34] [23] have been proposed and standardized for modeling and studying the behaviors of systems at different levels of formality and expressiveness. There is a wide range of formal and non-formal notations and their respective composition and verification mechanism presented in the literature for this purpose.

Glinz [18] uses Statecharts to model formal scenarios. He aims at modeling the relationship between the scenarios, regardless of their internal structure and to detect inconsistencies among them. His model of use case is based on Statecharts. His integration is based on the assumption that use cases and scenarios are modeled in disjoint contexts. If the designer come up with overlapping use cases, it should be treated as composing elements and therefore be decomposed into disjoint use cases. Over this disjoint use cases, he defines four constructors of sequencing, alternative, iteration and concurrency for constructing new use case models from the initial ones.

Based on this formal model, they present a new approach to object oriented modeling of software in their ADORA project [19] a modeling language which is based on the use of abstract objects. As an extension of this work, Ryser [35] introduces a new kind of chart and notation to model dependencies among scenarios, presenting a notation rather than a methodology.

Bordeleau et al. [12] [11] have proposed integration patterns for scenario dependencies. A state-based specification per use case is generated for each component and integrated to reflect the scenarios dependencies. However the whole process is done manually and relies on the creativity of the analyst to connect together the different Statecharts in the right way. They argue [13] for the systematic design of hierarchical finite state machines from scenarios. They do not assume that a component's behavior is necessarily typical, that is, it matches a particular behavioral pattern. Instead, they overview how to construct the behavior of a component from the scenarios in which it is used in order to assemble together the different roles this component plays. In this point, their perspective is informal and their specification generation process is non-automated in terms of composition.

Araujo et al. [9] focuses on representing aspects during the use case modeling. They propose to differentiate between aspectual and non-aspectual scenarios. Similar to our approach, the integration is done on the state machine level. The relationships between use cases are defined through an interaction pattern and defined in term of

roles. In our case, we define composition operators to generate new use cases that integrate the behaviors of the original ones.

Amyot et al. [6] presents a notation to model the main functional aspects of the system as well as their internal relations and their structure. In their approach, scenarios are described as causalities and responsibilities, allowing a higher level of abstraction than diagrams such as message sequence charts. They use Use Case Maps (UCM) for presenting their scenarios. A scenario is seen as a sequence of responsibilities (events and activities) that occur internally or externally, regrouping together in a casual manner, to serve as a certain functionality.

3.3 Scenario Composition

Alur et al. [2] presented an algorithm for checking the completeness of the message sequence chart specifications raised from the composition of MSC scenarios on an abstract notion of HMSC. The algorithm produces missing implied partial scenarios to help guide the designer in refining and extending the specification. It refers to the designer when the implied MSC is detected, and it generates automatically the implementing state machines from the MSCs for checking of the completeness of the composition. The technique can be applied on different communication architectures (asynchronous vs. synchronous, FIFO or non-FIFO message buffering...) [4] [2]. Their rigorous theoretical work lacks a convenience tool support to practice

the idea on large industrial specification problems. Moreover, the decidability and realizability of the MSC compositions has been widely studied [3] [5] [26] [27]. It has been shown that synchronous HMSCs are decidable in the P-complete order of time while the asynchronous HMSCs are decidable only in NP-complete space of time order. Moreover on MSCs, **Ben-Abdellah et al.** [10] addressed a very important issue of process divergence and non-local choice in the composition of MSCs in particular, and any scenario composition notation in general, and present a syntactic method to detect and resolve them with a tool support for practicing their approach on the MSC scenarios.

In a series of works from **Uchitel et al.** [43], [42], [44] address the importance of implied scenarios in the generation of behavioral model specification. They present a methodology and a series of algorithms for detecting implied scenarios emerged from the composition of MSC scenarios, and further use these scenarios for incrementally constructing an elaborated specification of negative and positive scenarios [45] [46]. They use their approach of detecting implied scenarios for determining if the generated scenario is an intended or unintended one, and hence the intended system specification evolves during several incrementation.

Whittle et al. [49] presented and algorithm and tool support for automatically generating UML Statecharts from the combination of UML Sequence Diagrams (SDs)

with a set of pre and post conditions given in UML OCL (Object Constraint Language). Their algorithm tackles three main issues related to generating statecharts from sequence diagrams. First, they detect and resolve conflicts arising from the merging of independently developed sequence diagrams. Secondly, different sequence diagrams with identical or similar behaviors are recognized and merged for a true interleaving of sequence diagrams. Finally, they introduce a hierarchy for statechart generation to make the generated statechart more structured and readable. Later, they study their approach on a large-scale specification of an air traffic control [50].

Some et al. [39] have developed an algorithm for the synthesis of parametric timed automata [1] from structural textual scenarios or extended MSCs. They present an incremental algorithm that merges scenarios with respect to timing constraints to obtain the desired behavior. They have further extended their work with operations semantics and conditions to define states in order to group states and hence, reduce the complexity of their algorithm [38]. More recently, Some introduced in [37] a new notion for the modeling of use case precedence in the specification generation process and their verification and validation. He presents the addition of use case description elements to explicitly express sequencing constraints between use cases. With complementary constructs to specify which use cases need to precede a use case and how these preceding use cases are synchronized. The second construct allows to specify which use cases are enabled from a use case and how these use cases execute concurrently.

More recently in [48], **Kicillof** and **Grieskamp** introduce a schema language for modeling and composition of behavioral models based on action machines, a derivation of label transition systems. They have algorithmic composition mechanism for each composition semantics separately and act on a lower level of semantics, closer to the design. However the semantics of composition operators in our approach in deployed in builders, hence a unique merging algorithm is used for all the existing and future operator.

3.4 Supporting Tools

In [28], **LTSA** is presented as a verification platform for state-based concurrent systems. It mechanically checks that the specification of a concurrent system that satisfies the properties required of its behavior. In addition, LTSA supports specification animation to facilitate interactive exploration of system behavior. A system in LTSA is modeled as a set of interacting finite state machines. The properties required of the system are also modeled as state machines. LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties. More formally, each component of a specification is described as a Labeled Transition System (LTS), which contains all the states a component may reach and all the transitions it may perform. However, explicit description of an LTS in terms of its states, set of action

labels and transition relation is cumbersome for other than small systems. The tool allows the LTS corresponding to a FSP specification to be viewed graphically.

LTSA is a useful tool with expendable features which can be used as a start point for the implementation of algorithms and approaches. As an extension plug-in, LTSA-MSC [41] provides the tool support for the method presented in [46, 43]. It supports the elaboration of behavior models and scenario-based specification by providing scenario editing, behavior model synthesis, and model checking for implied scenarios. It allows models to be described by graphically editing sets of scenarios in the form of message sequence charts. The LTSA-MSC can be used to detect the presence of implied scenarios in the system as part of an iterative design process. Furthermore, this analysis is used to incrementally generate the desired system specification [46].

As another extension to LTSA, **Foster et al.** [17] presented tool support for a model based approach to the composition and verification of web service implementations. Their approach [16] supports verification against specification models and assigns semantics to the behavior of implementation models to confirm expected results for both the designer and implementer. Specifications of the design are modeled in UML, in the form of Message Sequence Charts (MSCs), and automatically compiled into the Finite State Process notation (FSP) to concisely describe and reason about the concurrent programs. Their work is on the context of web services choreography. The LTSA-WS tool benefits useful features on the scenario design in MSC notations, as well as synthesis verification for the web service compositions.

Harel et al. [30] has proposed a methodology and tool support to model the requirements based on LSC, and studies the possibility of generating code from them. Their method is based on a play-in and play-out technique of the requirements presented in [21], and a BDD model checker is used in their tool to verify the actions done by the user and its consequences. It consists of a *play-in play-out* mechanism for the requirements as input LSCs [14]. The play-in concept is a high-level user-friendly graphical interface GUI that allows the user to capture the requirements of the target system by describing the sequencing of the model messages as desired by the user. The play-out however consists of executing the requirements interactively, without building the model or writing the code. The tool translates the play out task into the corresponding model, runs the model checker, and injects the counter example from the breach to the play engine for visualization. They end up with a system specification with no exhaustive verification over the behaviors and hence, some implied scenarios can remain undetected.

As an earlier work, **Alur et al.** [4] presented both a method and tool support illustrating that message sequence charts are open to a variety of semantic interpretations such as whether one allows or denies the possibility of message loss or message overtaking and on the particulars of the message queuing policy to be adopted. They present an analysis tool that can perform automatic checks on message sequence

charts and can alert the user to the existence of subtle design errors for any predefined or user-specified semantic interpretation of the chart. The tool can be used to specify temporal constraints on message delays and can then return useful additional timing informations such as the minimum and the maximum possible delays between pairs of events.

Mäkinen et al. [29] presented an algorithm, and a respective tool support named as MAS (Minimally Adequate Synthesizer) which synthesize UML Statechart diagrams from sequence diagrams in an interactive manner. The algorithm interacts with the user to guide the process in the critical points, hence lacks the automation in the composition. It keeps track of the interaction with the user, trying to learn from the desired behaviors and minimize the interactions. In addition, it detects inaccurate or incomplete sequence diagrams.

Calife [40] developed as an interfacing platform to allow the specification and formal validation of systems described as synchronized product of automata. In Calife, the goal is not to provide another verification tool, but to interface ideally all existing tools working on automata in a unique environment. Tools can be grouped using model definition associated with automata classes. The Calife platform works on several automata models (transition systems, timed automata, counter automata,...) and allows to define new models and interface new tools. They support the interface with some verification suites (UPPAAL, Hytech, Kronos, CMC, Coq,...). They also

support exporting the unique timed-automata system modeled under the Calife system editor to all these tools.

Several reasons have contributed for us to develop a whole new supporting tool instead of extending the existing ones. First, the UCOMV implements a novel approach with a unique model of use cases. The requirements for such model, with its composition mechanism [32], leads to specific interfacing features for the design and modeling procedure which are not found in other tools. For example, LTSA, with all its useful features on modeling and verifying features, would have to be largely customized to support our UCA model. Moreover, the verification mechanism in UCOMV which is based on temporal ordering of sequences is based on an XML based schema for transforming the model to a model checker. None of the above mentioned tools present such feature for their models. Calife supports many automata models. However the lack of documentation on their parsing method prevents us from customizing their tool for our approach. Other tools presented in this chapter such as [14, 29] and earlier [4] are all specific tools developed for the presentation of their approaches, rather than providing a platform for reasoning about other models. The implementation of our specific features need specific architectural designs which was hard to achieve in other existing tools. For example, retrieving hierarchical compositions needed specific data structure.

Chapter 4

Use Case Automata

4.1 Introduction

In this chapter, we present the theoretical work on which this thesis is based on. We present the Use Case Automata (UCA) as our formal behavioral model in the system specification. We also present the theoretical background needed for incrementally composing these partial system descriptions and gradually deriving the intended system specification. The basis of the theory on the modeling and composition has been originally developed and detailed in [31], and is rooted in our earlier research works of [32, 33] and [24]. Here in this chapter, we give a brief overview of the formal platform and principal structures needed for our approach.

The chapter is organized as follows. In Section 4.2, we present UCA as the basic behavior model used in our approach. Section 4.3 provides an overall description of

the composition approach, its steps, and its artifacts. It also defined how the overall system specification is gradually incremented after the evaluation of each composition expression list. Steps of the composition and the merging algorithm as well as the refinement process are presented in Section 4.4. Finally in Section 4.5 different composition semantics are presented and the available operators are introduced. Section 8.7 concludes the chapter.

4.2 The UCA Model

In our approach, a partial system behavior is modeled as a Use Case Automata (UCA). UCA is structurally an automaton. Each scenario is presented as a word of this automaton and the set of these scenarios form the language of the UCA as partial descriptions of system usage. Each UCA model acts as a partial functionality of the system which models the interaction between the system and its environment. The model acts as the reference for a behavior model and is be instantiated for the use in the composition and generation of new UCA. Hence formally, a *Use Case Automaton* is a 5-tuple structure (S, s^0, S^f, L, E) where S is the set of states, s^0 the initial state and S^f the set of final states, L is an alphabet and $E \subset S \times L \times S$ is the transition function, defining the set of transitions.

Literally a UCA is an automaton modeling partial interactions between the system and its environment as sequences of events. Each UCA carries a set of traces

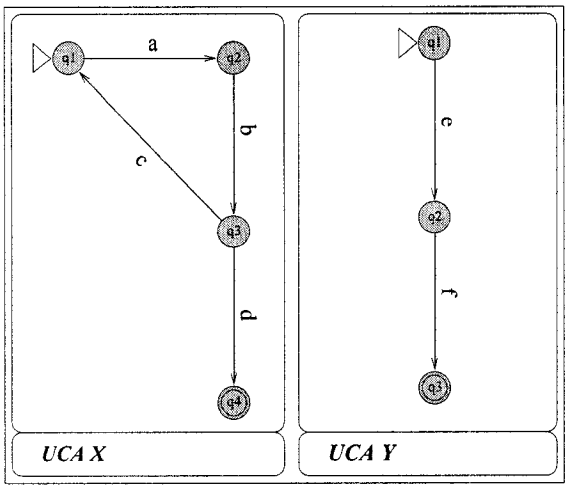

Figure 2: Example UCAs

as its scenarios which form the language of the automaton. The set of states reflects the conditions that the system could rest after reading the certain set of sequential events. Moreover the alphabets of the UCA reflects the set of possible events that the behavior model can interact on, and the reflection of these labels on the transitions constructs the executions of the model. Through the set of the alphabets of the UCA, any ordering of sequence of input alphabets is possible as the input to the UCA. However only sequences acceptable to the language of the UCA would be considered as a UCA scenario and would act as a execution of the system functionality. Figure 2 shows two examples of UCAs.

As the extension to the UCA model, the *Extended UCA* is introduced as enrichment of the UCA model with variables and guard conditions. The extended model would be the decorated UCA model with variables, condition on transition firing, and the assignments which are performed after firing the transitions. The formal

definition of the extended UCA is also detailed in [31].

Hence, an *Extended Use Case Automaton* (E-UCA) is a 7-tuple of the form $(S, s^0, S^f, L, V, I, E)$ such that S is the set of states, s^0 in the initial state, $S^f \subseteq S$ is the set of final states, L is the set of labels, V is the set of variables, $I \subseteq V$ is the set of input variables, $E \subseteq S \times C \times L \times A \times S$ is the transition relation such that C groups the set of pre-conditions on variables, and A the set of variable assignments. The set of pre-conditions are to be true before the transition is fired, and the variable assignments play the role of a post-condition. The set of variables given in the definition of an extended UCA are local variables to that UCA, and are valid through the scope of that UCA. Each variable should be unique in its name through its scope. According to the extended UCA model, the behaviors of the model and its scenario traces are also extended with variable values. This means that a set of execution scenarios of the EUCA is enriched with variables and their assigned values. EUCA in our approach is deployed in the same way as the UCA. Therefore in the rest of this chapter, we speak of the UCA and EUCA as the same notion following the same approach.

Clones

Each UCA, or EUCA, can be used to generate a more compound behavior. Each couple of UCAs is used to compose new behaviors through a formal automated composition. However, the generation of the new behaviors does not constraint the original UCAs. The composition of new behaviors would form an instance of the original UCA, called *clone* of that UCA. The clone of a UCA is generated using a labeling function acting on the transition labels. These clones are further used for composition of new UCAs, acting as new behaviors for the system. More formally, the clone of a UCA $A = (S, s^0, S^f, L, E)$ respecting a renaming function $Rename : L \rightarrow L'$ is a UCA $A' = (S, s^0, S^f, L', E')$ such that[1]:

$$\frac{q = (s_1, l, s_2) \in E}{q' = (s_1, Rename(l), s_2) \in E'}$$

Literally, clone of a UCA is renamed copy of a UCA behavior model. The transition labels of this copy are renamed according to the use of the UCA in the composition. Therefore, for each two clones, it is guaranteed that they would have disjoint sets of transition labels during the composition. Based on the number of places where the composition is to be done, several clones of a UCA may be needed for composing a new UCA. Here in the examples of this chapter, we just present a composition with one clone for simplicity. Figure 3 shows the clones for the two given example UCAs in Figure 2. You can see in the examples that how transition labels are decorated with the renamed labels according to their UCA. For more details on the UCAs and

[1]The $\frac{x}{y}$ notation is a logical implication notion, meaning that the upper expression would imply the lower expression. For example $\frac{p}{q}$ implies that if p is true, therefore the q would be logically true.

their clone generation, please refer to [33].

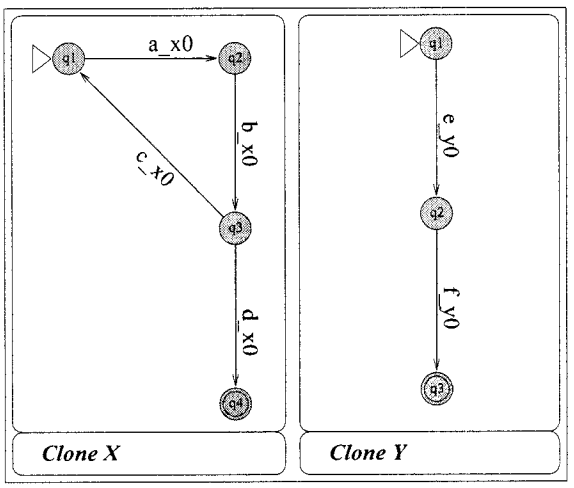

Figure 3: Clones of the two example UCAs

4.3 Approach Overview

Our composition approach, illustrated in Figure 4 and fully described in [32] and [33], defines the theoretical base for formal modeling and composition of UCAs and Extended UCAs as our behavior models. The designer starts with defining the preliminary specification with the first set of UCAs and a list of composition expressions for the generation of new UCAs from the existing ones. In each increment, a list of these composition expressions are evaluated and new behaviors are composed and added to the exiting UCA set. Each composition expression determines information needed for generating a new behavior model. The information includes two UCAs assigned for behavioral composition, named as *base* and *referred* UCAs, a composition operator,

and a set of *extension points* which gives the exact places in the base UCA where the new behavior should be inserted. Extension points could be expressed as states of the base UCA or transitions preceded by *Before* and *After* qualifiers. Thus, the behavior of the new UCA would be formed from the two existing behaviors according to the compositional semantic of the operator within the proper extension points.

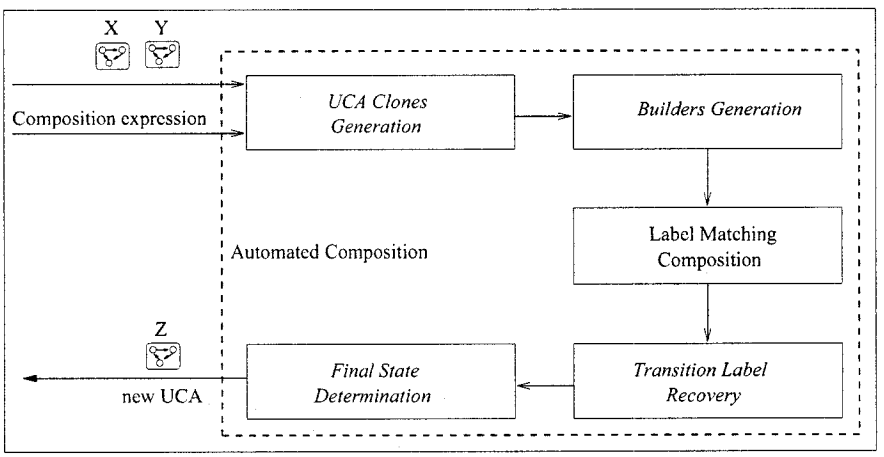

Figure 4: Composition Approach[32]

The evaluation of a composition expression consists of several steps. First, clones of the UCAs are generated as instantiations of the UCA model following a renaming function, as described before. From each clone, *builders* are constructed according to rules defined for the semantics of the expression. A builder is a UCA generated from the involving base or referred UCA, carrying additional transitions and states added to reflect the compositional semantics of the composition expression. They serve as inputs to the composition algorithm. The two builders generated from the base and referred UCAs are called *base* and *referred* builders respectively. Base builder reflects

the semantics of the composition operator in the extension point of the base UCA, as the referred builder prepares the referred UCA for the composition. Two labeled transitions of `begin` and `end` are added at the extension point of the builders which shows the beginning and ending of the insertion of the referred behavior in the base one. These two transitions will serve as common labels in the merging algorithm which is based on the label matching mechanism. It is notable that the intersection of the two alphabets of the involving builders should exactly be these two added transition labels. The proper insertion of these two transitions in the extension point would guarantee the composition of the traces according to the intended compositional semantics of the operator. These transitions are further removed from the composed automaton through a removal algorithm. This algorithm in turn, guarantees the removal of the added `begin` and `end` from the scenarios of the UCAs, maintaining the original scenario set of the composed UCA. The formal details of the rules on how builders are generated and composed are given in [33] and [31]. Figure 5 shows an example of two simple UCAs and the expected model from their composition.

Evaluation of a list of composition expressions would form an evolution in the specification by adding the newly composed UCA models. Therefore, starting from an initial set, further new UCAs can gradually form the desired specification with intended behaviors, which is subject to behavioral verification for desired properties.

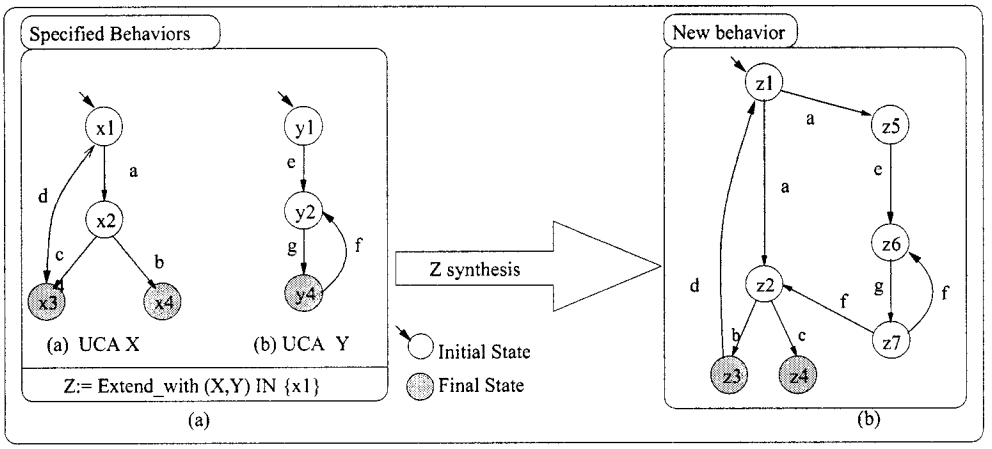

Figure 5: Expected UCA from evaluation of an expression

As for the case in the extended UCA, the same methodology is deployed for incremental generation of the specification. The composition is expressed using the extended version of the composition expressions. However the composition is performed using the same approach of generating builders and following the same merging and removal algorithm, decorated with variables on transitions.

4.4 Composition

In our approach, the composition is the insertion of the behaviors of the referred UCA in the traces of the base UCA according to the compositional semantics of the operators. These traces are composed on the *extension points* inside the base UCA, which is specified in the expression of the composition along with the two UCAs.

Extension points can be consist of several states or transitions. Accordingly, the *Extension Set* is defined as a non-empty subset of states or transitions from the base

UCA. For the given composition examples presented in this chapter, we restrict for simplicity reason to composition expressions with single extension points for simplicity. Hence the extension sets in our examples has one element as a state or a transition. Generic rules and details for several extension points can be found in [32] and [33].

The composition on the extension point is performed according to the composition semantics of the operator used in the expression. We define composition operators as binary operators between two UCA, carrying certain semantics for the composition of the traces of two UCAs. We have some predefined operators such as Include, Extend, and Alternative. The supporting tool, as it will be discussed in Chapter 5, provides a ground for the description of the new operators. Evaluation of a composition expression results in the generation of a new UCA. Hence, the new UCA is expected to reflect the set of interleaving traces of the two UCA according to the semantics given in the composition operator in the extension point.

Builders

The semantics of the composition operator for the composition is implemented using builders. Builders are automata built from the two base and referred UCAs, reflecting this compositional semantic on the extension point. The two base and referred builders act as the inputs to the merging algorithm, which will finally result in the composition of the new UCA model. The base builder is generated from the base

UCA from the operator's compositional semantics on the extension point, while the referred builder is based on the referred UCA and is synthesized independently from the compositional expression.

Each base builder is generated from the base UCA in the UCA expression reflecting the traces in the base UCA enriched with two labels of *begin* and *end*. The structure of this builder is constraint by the rules defined according to each composition operator and the extension point specified in the expression. Here we present an example of two builders. For formal details and more examples of the builders please refer to the respecting research works [32, 33]. Figure 6 shows an example of a base builder generated for the example UCA X given in Figure 2. You can see how the `begin` and `end` transitions are added according to the semantic of the Include operator in the extension point $s = q_2$ in the UCA X. These two transitions mark the beginning and ending of the composition in the base UCA. These two transitions would act as the common base on out merging algorithm. The combination of the three mentioned operators with different types of extension points as states or transitions leaves us with 16 distinct semantics for the generation of base builders which can be used in the UCA expressions. The formal details of all these builders are given in [33].

The second builder used for the composition generated from the referred UCA and is called *referred* builder. It is simply the referred UCA with added transition of *begin* and *end*. Hence the generation of the referred builder is independent from the

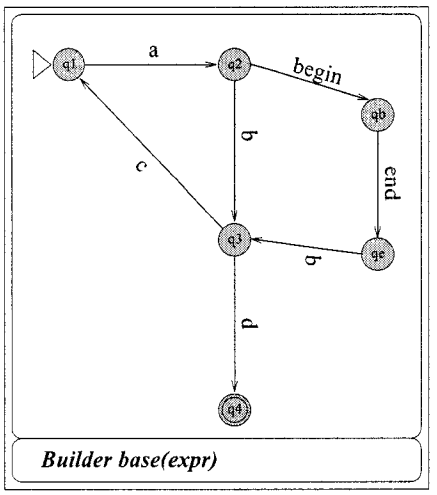

Figure 6: Builder example for UCA X

semantic of the composition operator and the extension point in the expression. The generation of the referred builder is adding an initial state to the original UCA, and adding the begin and end transitions to prepare the UCA for the inclusion inside the base UCA.

Merging

The two builders are then used for the composition and generation of the new UCA. This is done through the merging algorithm which is based on the label matching mechanism between the two automaton. The label matching algorithm define a sort of synchronization on message labels of transitions of the two automata. If the transition label is a common label of the two alphabets, the algorithm fires a transition with that common label on the new automaton. Otherwise, each transition label is defined separately. The basic form of the algorithm works for two automaton, however the

extension of the algorithm for n automaton [33] is given in Definition 1. We once more note that the only shared labels between the builders is the two added labels of begin and end, marking the beginning and ending of the insertion of the UCAs.

Definition 1. *(Label Matching).* Let $\{A_1, A_2, ..., A_n\}$ be a set of UCA where $A_i = (S_i, s_i^0, S_f^f, L_i, E_i)$. Their behavioral composition, denoted as $A_1 \bullet A_2 \bullet ... \bullet A_n$ is a UCA $A = (S, s^0, S^f, L, E)$ where $S \subset S_1 \times ... \times S_n$, $s^0 = (s_1^0, ..., s_n^0) \in S$, $S^f \subseteq (S_1^f \times ... \times S_n)$, $L = L_1 \cup L_2 \cup ... \cup L_n$ and E is the smallest relation in $S \times L \times S$ that satisfies the following rules:

$$\frac{(s_i \rightarrow^l s_i' \in E_i, i \in J) \; where \; J = \cup\{i | s_i \rightarrow^l s_i' \in E_i\}}{((s_1, ..., s_n) \rightarrow^l (s_1', ..., s_n') \in E) \mid (s_i' = s_i \; if \; (i \notin J))}$$

The resulting automaton from the merging algorithm is an intermediate automaton, which contains the original interleaved traces of the two UCAs as well as begin and end transitions. The automaton needs a final process to become a UCA and could be added as a new behavior model to the specification.

The final process in fact consists of two major procedures. First the transition labels which which are renamed and indexed during the cloning process should be converted to their original form. This would bring the transitions to their original

labels as they were presented in the UCA alphabet. Second, the two added transitions should be removed following a removal algorithm which acts on the transitions carrying these labels.

This algorithm is based on the classical lambda removal algorithm. The two begin and end transitions are treated as lambda transitions and are subjected to act as an empty message event. However we decided not to use the classical lambda transition removal algorithm [25]. Instead, we define a customized version of the lambda removal algorithm which performs the elimination of the transitions on some specific transitions. This is because of the fact that the original lambda removal algorithm surfs on the whole automaton to detect the lambda labeled transitions. However, we have the specific privilege on knowing exactly which transitions would carry such labels a priori. We know that we only have two transitions to be deleted, and moreover, we know the exact transitions to act on. Therefore we use these information to develop our own customized removal algorithm and reduce the complexity of surfing the whole transition set of the automaton to the linear complexity of removing two transitions. Figure 7 shows the result of the evaluation of our example composition expression from the UCA X and Y before and after the refining process.

4.5 Operators

In our approach, composition operators are binary operators each defining a unique semantic for the composition of UCAs. These semantics reflects how the traces of the

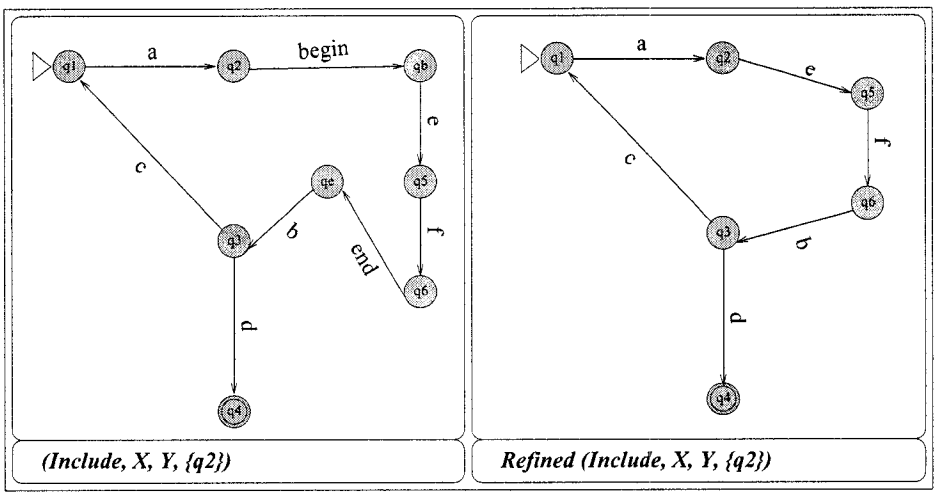

Figure 7: Resulting automaton from merging and refinement

two involving UCAs are composed together and the trace set of new UCA model is formed. Each operator carries formal rules for the deduction of the new UCA trace set. These rules define the set of the new behavior according to the sets of the involving base and referred UCAs. These rules act on the affecting subset of the trace set of the base UCA determined by the definition of the extension point in the expression. As an example, we give the formal rules of trace composition of basic operator Include reflecting its compositional semantics. Assuming A_1 and A_2 as the base and referred UCAs respectively, *ep* as an extension point in A_1, and $L(A_1.ep) \subset L(A_1)$ as the subset of traces passing through *ep*, the building set of *opr* reflects the set of traces of the new UCA.

$$\frac{w_1 = u.v \in L(A_1.ep), w_2 \in L(A_2)}{u.w_2.v \in opr}$$

$$\frac{w_1 \in L(A_1) - L(A_1.ep)}{w_1 \in opr}$$

These rules are applied for the composition of traces of the two UCAs and the resulting set of these rules would be the deduced trace set of the new behavior from the composition expression. The abstraction of these rules and their compositional capacity defines a logic on the composition of traces and rules for the deduction of new behavior sets from the existing ones. These rules could be overridden for introducing a new compositional semantic for a new elaboration of the trace set of the composed UCA. As can be seen in the example of Include, definition of these rules include how the traces from the base trace set are determined, and moreover, how they are composed with the traces of the referred UCA and further deployed in the new trace set. Any intended composition semantic on the traces can be expressed using this logic and the refinement of these semantical rules. Each operator carries a set of these rules which are implemented and applied for the composition of the new UCAs.

Extension Point

The composition expression determines a subset of traces in the base UCA that are affected by the composition, and not necessarily the whole set of traces. This subset is determined by the extension point. As explained before, extension points are the places inside a UCA where the compositional semantics of the operators are to be reflected. The set of extension points in each composition expression can include states or transitions. It can be specified through explicit states and transitions, or a characterization of transitions inside the UCA in terms of transitions carrying a certain label. In both cases, each extension point partitions the set to two subsets

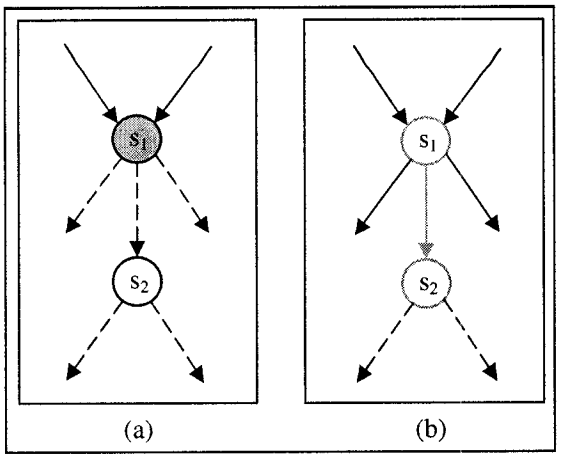

Figure 8: State and transition extension points and their affected traces

depending on if the trace would be involved in the composition or not. The extension point reflects the exact point of the sequence within the trace in which the new traces are to be composed. Depending on the type of the extension point (state or transition), these traces can be traces passing a certain transition, or all the traces passing a certain state. Generally speaking, the cardinality of the set of traces determined by a state is bigger that the one determined by a transition. Figure 8 shows examples of two types of extension points and their respective affected traces.

Defining Builders

New operators for the UCA composition can be created by overwriting the compositional rules of the trace composition. In our approach [32], the composition semantics of the operators are implemented through the notion of builders. As mentioned before, builders are finite state automata which are built from the two involving UCAs, reflecting the proper semantic on the extension points for the composition. Builders

derived from the base and referred UCAs are called *base* and *referred* builders respectively. These two builders would be fed as the inputs to the merging algorithm of composition which result in an intermediate automaton, further processed for the generation of the new UCA.

UCOMV provides a platform for the introduction of new compositional semantics. It provides a platform for defining new operators and their respective builder templates. The analyst can define new builders and reflect their intended semantics regarding the base UCA in the composition. The introduction of the new operators would consists of defining proper composition rules for the traces and implementing them on proper classes. The implementation of these rules are done on the classes encapsulating the construction of the base builders. They contain the structural information needed for the generation of the builders. This information includes the states added to the base UCA, the added transitions, two of which are the begin and end for the marking of the exact place of composition in the UCA, and how these added transitions would reflect the semantic on the UCA. An example of these builders for the it Include operator is given in figure 9. This example shows the conceptual semantic of the Include and how the composition rules of the operator would be reflected on the base builder (a), and how this would result in the intended trace inclusion of the referred UCA inside the base traces. Dark states in the generic builder marks the extension point. As you can see in the figure, Include is an operator which the reflection of its semantic needs only one extension point. Other possible operators

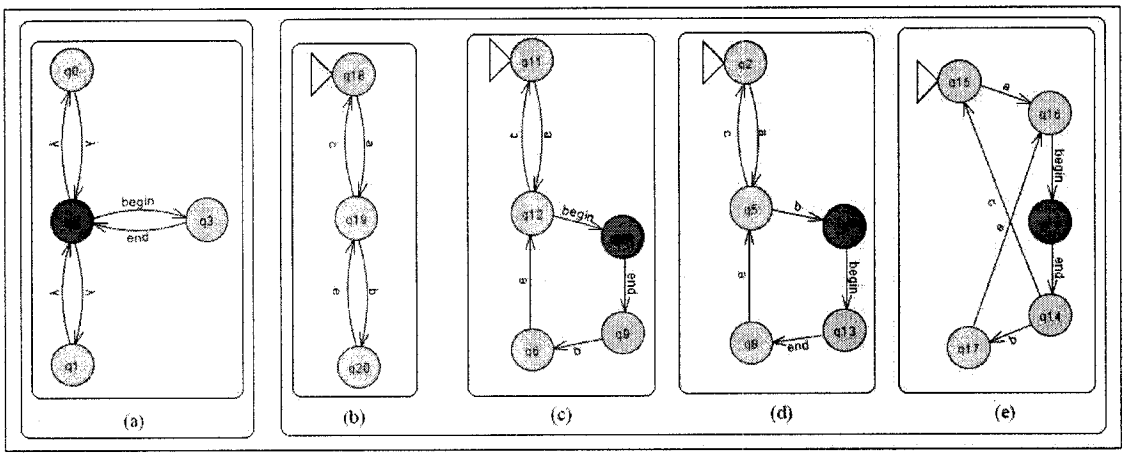

Figure 9: Concept of Include builder (a), the example UCA X (b), and the generated builders before and after transition b (c),(d), and on state (e)

might need more extension points to implement their semantics. Figure 9 shows how the generic concept of the builder of Include would be used for the construction of base builders for a given UCA .

Current Operators

Several operators have been implemented so far for the most used semantics in the composition of behaviors. These include six operators, named as Include, Extend, Alternative, Refine, Graft and Interrupt. As of Include, the behavior of the referred use case is inserted in the behavior of the base use case in the extension points, while the Extend operator aims at inserting the behavior of the referred use case in the traces of the behavior of the base use case as a possible alternative in the extension points. Alternative acts as an alternative for the behavior of the referred UCA to the behavior of the base UCA in the extension points. In Refine, the extension point is refined

with the behavior of the referred UCA, meaning to replace behavior in the extension point with the whole behavior of the referred UCA. Graft act as an extension of Extend which can start in and extension point and end in another distinct one, and finally, Interrupt is the extension of the Alternative on all the states possible in the base UCA. The detailed semantics of these six operators is given in [32]. The two types of extension points as states or transitions, as well as the Before and After qualifiers on the transitions, combined with our six pre-defined operators provides eighteen distinct semantics for the composition of traces in the tool. Proper base builders would reflect these semantics on the traces of the base UCA during the composition. This abstraction of the operators provides a common ground for deriving user-defined compositional semantics, and gives the ability to the analyst to produce more optimal UCA specifications using his customized operators. Figure 10 delineates a picture of how the compositional semantics are performed for three operators of Include, Extend, Alternative.

4.6 Conclusion

In this chapter, we presented the theoretical framework for the modeling and composition approach deployed in this thesis. We presented UCAs and the EUCA as our formal models of behavior, and discussed the approach of the composition of new behaviors. We also discussed our customized removal algorithm and the reason we deployed such algorithm. Finally we discussed the composition semantics of the

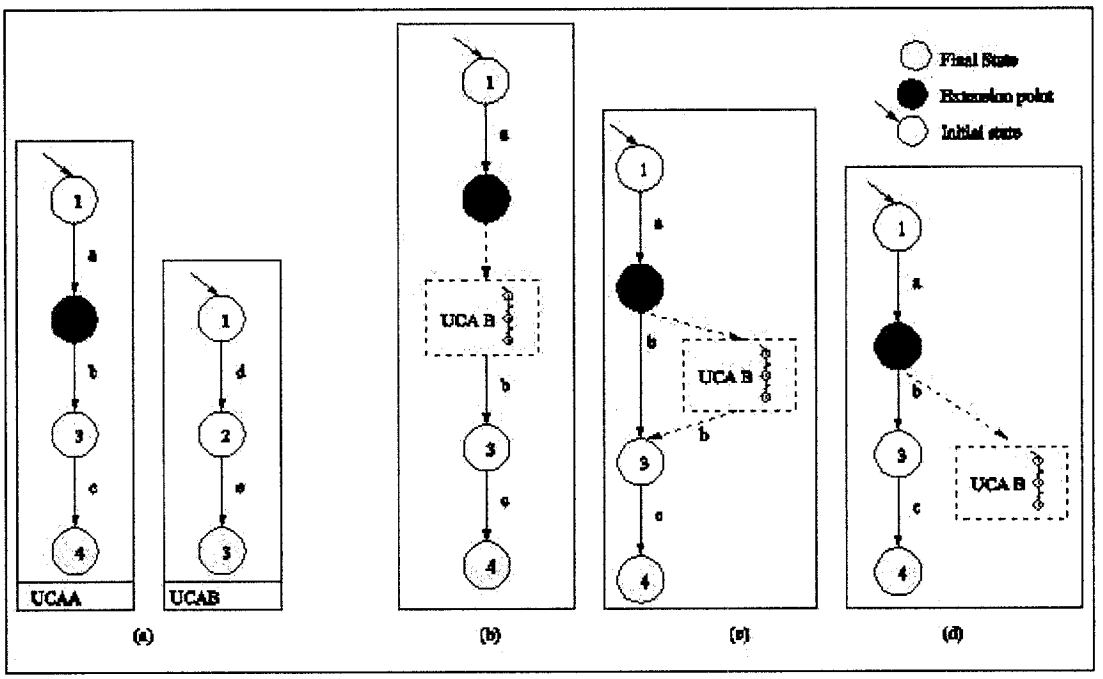

Figure 10: Overview of the operators composition semantics [33]

operators and introduced the currently operational operators in the tool. In the next chapter of the thesis, we study the tool we have developed for the modeling and composition approach presented here. In the next chapter, we introduce the tool support for practicing this approach on specification generation problems.

Chapter 5

UCOMV

5.1 Introduction

Despite their proved advantages, formal models of system requirements are usually hard to work and understand, and needs a level of mathematical expertise from the stakeholders. Providing tool supports can ease this process and help the analyst for an easier study of their prospective system, and the stakeholders to better understand the intended system, benefiting from a rigorous formal behavioral analysis.

Based on the theory of composition presented in Chapter 4, we have developed a tool and provide a platform to support practicing the approach on the real-life specification problems. The tool implements the modeling and composition theoretical work presented in the previous chapter, and propose a strategy for incremental generation of system specification from initial basic requirements.

The chapter is organized as follows. The properties of the tool and the idea of the development of an integrated framework are presented in Section 5.2. Internal design of the tool architecture has been discussed in Section 5.2.1 and different parts of it are reviewed. It is also discussed why this architecture has been chosen. Different layers of the tool are introduced in Section 5.2.2, each of which is described in their design and feature details in their respective chapters. Section 5.3 concludes the chapter.

5.2 Integrated Framework

The composition theory presented in previous chapter was the original motivation for the development of a supporting tool. The idea was to develop a tool make it possible to practice the application of the modeling and composition theory on real-life specification problems. However, the original idea evolved into an integrated framework for formal modeling, composition and verification with many interesting features on the improvements in the theory.

Within the presented tool an incremental specification generation strategy is proposed and implemented. According to this approach, the target system specification is modeled and verified in incremental steps. Basic functionalities of the system are introduced as UCAs and EUCAs, composed to generate new behaviors, verified over certain behavioral properties, and gradually form the target specification with

desired functionalities. This integrated framework provides a solid ground for incremental elaboration of system specifications, and can help the analysts for a better study of the prospective systems.

The tool is called **UCOMV**, standing for **Usecase Composition, Modeling and Verification** tool. It has been designed and implemented in Java, with full object-oriented features on its various different sections. Its object-oriented design along with it Java implementation provides a groundwork for its future extensions, along with possibilities of collaboration with other extensions to the suite, such as other automata design modules and possible composition and verification interfaces.

5.2.1 Tool Architecture

As for the formal study of our behavioral model, three major steps of modeling, composition and verification has been seen for each incrementation of the specification. In UCOMV, as for the formal study of our behavioral model, three major steps of modeling, composition and verification are implemented in a layer. Features for the design of the specification are implemented in *specification interface* layer. This layer includes the data structures and features for interfacing the formal modeling of the behaviors through UCAs and EUCAs, defining composition expressions, and tracking their composition history.

Moreover, the composition theory presented in previous chapter has been implemented in the *composition engine* layer of the tool. This layer include the implementation of all the data structures and algorithms needed for performing our composition theory. It includes the core components implementing the steps of the composition from the builder generation, merging algorithm and transition removal process.

The third layer of the tool provide the implementation of a model verification approach on our approach. It presents the integration of a model checking tool to our framework and deploys a customization of the model verification approach to our modeling and composition methodology. Next section would detail the design of UCOMV and its components in more details.

5.2.2 Three-Layered Design

As discussed, UCOMV is implemented in three layers. Each of these layers is in charge of providing features for deploying the specification generation approach on real-life problems. Each layer along with is consisting components provides the implementation of the theory of the approach, with many useful features which will be described in details in their respective chapters. The layers are implemented with distinct interfaces to each other which would ease future extensions of the platform. Figure 11 illustrates the three layers of the tool and their corresponding components. It also shows how these layers communicate with each others through their interfaces. For each layer in the tool, different components has been implemented, some of which

are to support the formal theory of the related features, while others are to provide additional features to facilitate the process in that layer.

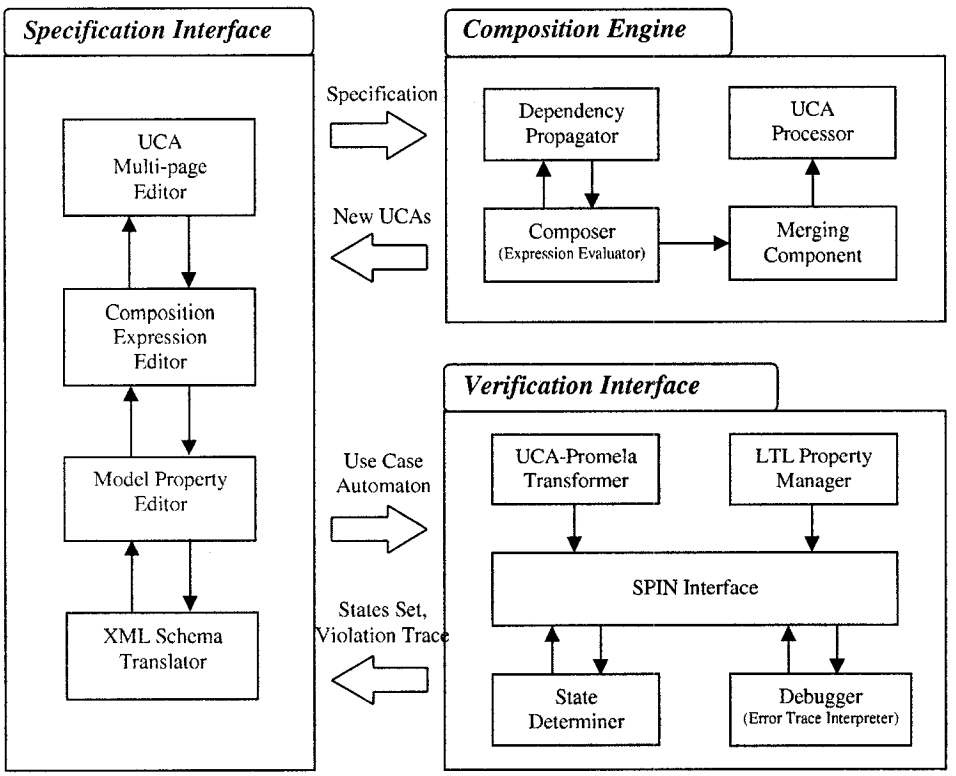

Figure 11: Tool Component Architecture

In Figure 11, each arrow shows the artifact that the module sends to the target module and gets the result of the process furthermore. For example the specification is interfaced in the specification interface module of the tool. The specification resulting from the interface includes a set of use cases and the list of composition expressions. This specification would be provided to the composition engine of the tool for the evaluation of the composition expression and generation of new UCAs and EUCAs. In the composition engine, the composition expressions are evaluated, and new UCAs

are generated. New UCAs are then given back to the specification interface to be added to the original specification and form an increment in the specification. These UCAs are then given to the verification module of the tool to verify the UCA over desired behaviors. Such interactions between the layers are modeled through wide arrows in Figure 11. Each wide arrow shows an artifact of one layer, being transfered to another one for further processed to be performed on.

5.3 Conclusion

In this chapter, we presented the integrated platform we have developed for the practice of our specification generation methodology on real-life specification problems. We presented the overall architecture of the tool and discussed how a three layered architecture design would contribute to the implementation, understanding of the implementation, and future extendability of the tool. It would also ease the interactivity between layers during the revisions made on each increment. In the next chapter, we present and discuss the specification layer as the first layer of the tool which encapsulate the specification generation components of the framework.

Chapter 6

Specification Interface

6.1 Introduction

The specification layer of the tool allows the user to define the specification of the system under development. As described in Chapter 4, each specification in our approach consists of a set of UCAs, a list of composition expressions, and a hierarchy of the composition history. The specification layer of the tool provides the proper data structure and interfacing feature for the design and management of such specification, such as automata design editor, composition expression presentation and management environment and transformer structure for mapping the UCA and EUCA models into verification module. This chapter describes different components and sections of the layer and discuss the design of the implementation in the tool.

The chapter is organized as follows. The automata design interface of the tool is

discussed in Section 6.3. The environment for defining and managing different types of composition expressions are detailed in Section 6.4. And the interface provided for defining and verifying properties on model checkers is discussed in Section 6.5. The semantic of the variables implemented in the tool to be used in EUCAs and their composition are discussed in Section 6.6. An interesting feature of the tool for coloring of states in UCAs and EUCAs is presented in Section 6.7 and finally, the XML schema developed in the tool for the presentation of UCAs, EUCAs and the specification is described in Section 6.8. Section 6.9 concludes the chapter.

6.2 Interfacing the Specification

The specification layer of the tool provides the framework for developing and managing specification as the set of UCAs and EUCAs, a list of composition and extended composition expressions, and a hierarchy of the compositions performed before in the specification.

The set of UCAs and EUCAs are presented and managed in an ordered list on the left corner of the tool. It gives designer the ability to add new models, view and edit existing ones, trace the hierarchy of a UCA or EUCA in the specification and change the default color of the states of the UCAs. Figure 12 shows a screen-shot of the UCA-EUCA management environment of the UCOMV. The environment also provides features for exporting and importing of separate UCA and EUCAs in specific

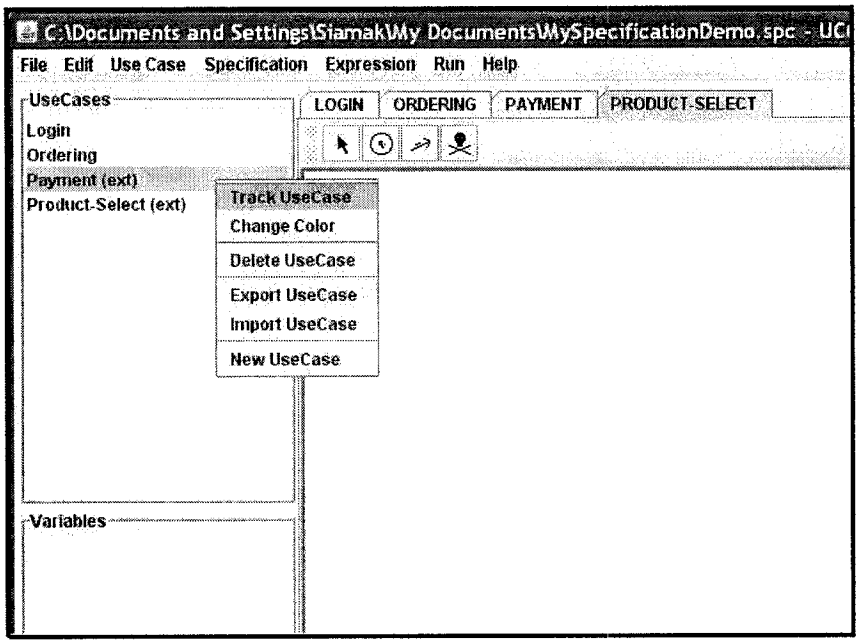

Figure 12: UCA-EUCA management environment

files. This feature enables the designer to export its common behavior models and import them in other specifications, providing the ability to produce a pool of handy behavioral models for future use. The exported file is an XML file generated with the specific XML schema implemented in the tool which retrieves the UCA and EUCA models into proper XML structures. Section 6.8 discuss this XML schema in more details.

An other management environment is implemented for defining and managing the composition expressions in the specification. This environment also provides features for defining different types of composition expressions and extended composition expressions. It also provides features for monitoring and editing the current expressions which are not yet evaluated and composed. Section 6.4 will also describe more about the features of this environment.

6.3 Automata Design Environment

The behavioral modeling in UCOMV is based on the mathematical model of the finite state automata. UCAs and EUCAs are both based on the structure of an automaton, with finite states, transitions and final states. For the support of this modeling approach in UCOMV, we have implemented a library with proper data structures for restoring the formal models of UCA and EUCA. In the specification layer of the tool, we have implemented an interactive environment for the graphical design of the UCAs and EUCAs and restore the proper data structure from it. This is the editor for the design of the structure of an (extended) automata with all its formal details.

The graphical presentation of the UCAs are developed with the help of the JFLAP [36] library. JFLAP is a tool for drawing automata and it provides useful features for drawing and editing automata in a graphical interface. For the purpose in UCOMV, the JFLAP drawing library is deployed within the *multi-page editor*, and is customized for the fit of the specific features in our UCA model. The classes of the library which has been used in UCOMV include the classes for drawing the automata and retrieving proper data structure from the visual editor. The graphical presentation of the JFLAP for simple automata has been enriched to support our Extended UCA model with variables and guard conditions. The classes which have been used from the JFLAP has been marked in the code with the signature they asked for. The multi-page editor in UCOMV provides full features needed for defining a complete

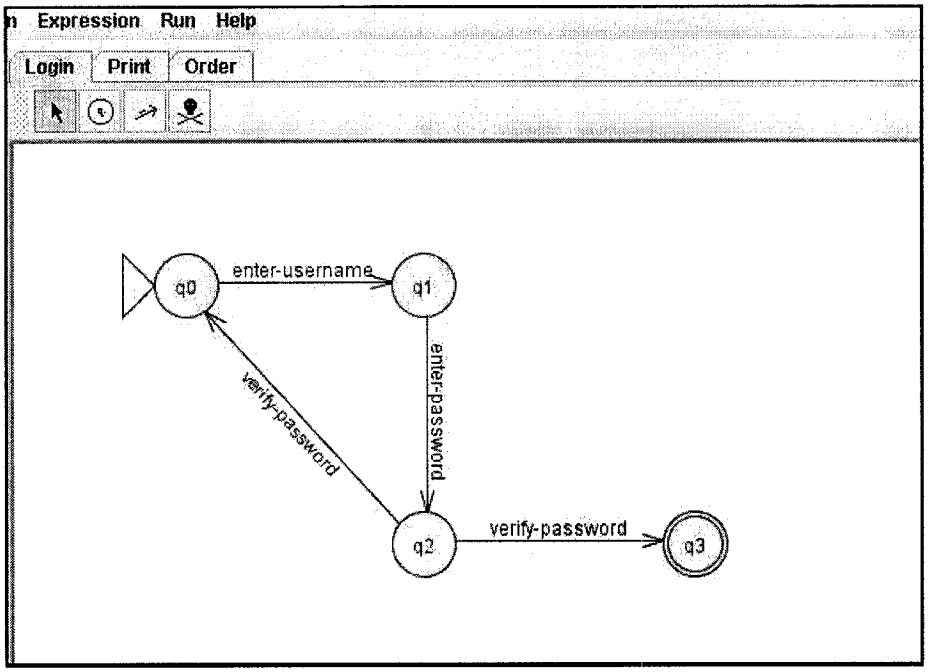

Figure 13: Automata Design Multi-Page Editor

mathematical model of automata. The analyst can draw, remove and edit the states, the transitions between states, the initial and final states, and additional state labels for better understanding of the UCA behavior.

The GUI is designed on a **Model-View-Controller (MVC)** architecture design. This means that the GUI design of the automata is handled separately from the original data structure used as the UCA model. The controller module in the interface is in charge of coupling the designed automaton with the UCA model. It also handles the structural changes performed during the design of the automaton. A structural change in the automaton include the change in the states, transition, the choose of initial state and the set of final states. As any of such changes happening to

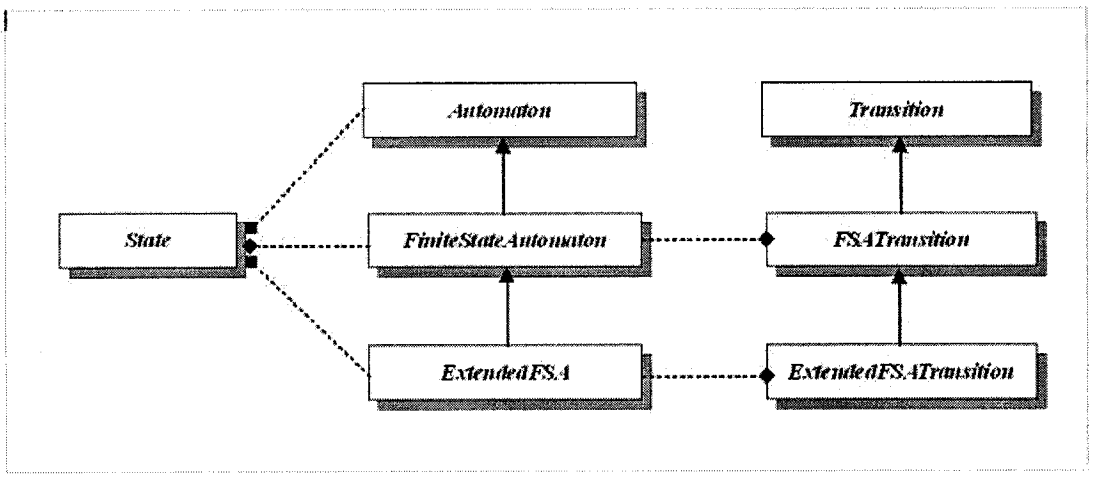

Figure 14: Overview of UCA and EUCA data structures

a UCA-EUCA, the controller listens to the change and catch the change in a parsing from the graphical model. Therefore, the UCA structural model is in a synchronization state during the design of the UCA model. The UCA code is generated from the designed graphical structure for the use in the composition and model checking phases. Originally the graphical model is encapsulated in a Java class representing the UCA model. Furthermore this structure is transformed to an XML based schema which is defined in UCOMV for this purpose. The XML code of the UCA based on the schema will serve as the code of our formal model of behavior for the further steps in the composition and verification.

The data structure preserving the formal model of UCA and EUCA are a hierarchy derived from the class Automaton. This base class encapsulates the basic structure of an automaton. It includes a finite set of class State and a finite set of transitions from the proper types. As you can see in figure 14, the class FiniteStateAutomaton is

derived from this base class and the class ExtendedFSA is in turn derived from that class. The FiniteStateAutomaton class contains a set of FSATransition, a class for encapsulating the structure of a FSA transition which is derived from the base class of Transition. Further the class ExtendedFSATransition is the derived class from the FSATransition, inheriting the same structure and enriching it with guard conditions for transitions. Each ExtendedFSA class holds a set of ExtendedFSATransition objects for preserving its extended transitions.

6.4 Composition Expression Design

As a part of the specification, a list of composition expressions is to be defined through the *composition expression editor* component of the specification layer. Each of these expressions is a description of the synthesis a new UCA model from the existing ones, composed through a compositional semantic from the operator on the extension point. Therefore the involving UCAs as well as the composing operator and the extension points are encapsulated within the composition expression in the specification. Along with its theoretical counterpart, the data structure representing the composition expression consists of two use case automata as the base and referred UCAs, a composition operator containing the compositional semantic of the expression, and an extension point which reflects the place of extension in the base UCA. The composition expressions in UCOMV are presented in the following syntax [32]:

Z := Composition_Operator (X, Y) Extension_Point

For the EUCAs, the extension composition expressions should be defined by the designer. The extended composition expression is an extension to the base composition expression which enrich it with conditions on the compositions and assignments for post composition valuations. The syntax for the extended composition expression is also the extension of the previous syntax with a composition condition, and two sets of input and output variable assignments and would be represented by the following syntax in UCOMV:

Z := Composition_Operator (X, Y) Extension_Point

Composition_Condition Input_Var_Assign Output_Var_Assgn

In the syntax of the extended composition expression, the composition condition is a set of conditions on base EUCA variables which should be validates before the execution of the referred EUCA in the composed model. Furthermore, the input variable assignment is a set of variable assignments which should be performed before the execution of the referred behavior while the output variable assignment is the set of such assignments being performed after leaving the referred EUCA in the composed model. The set of input variable assignments act as a sort of parameter passing for the insertion of the new behavior, and the output variable assignment plays the role of returning values from the referred behavior. Figure 15 shows the derivation of

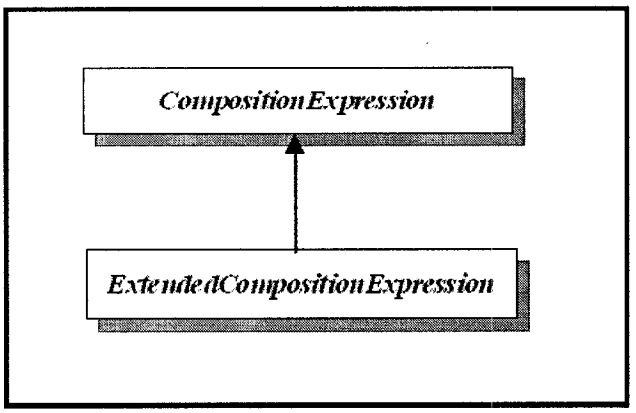

Figure 15: Derivation of composition expression classes

the class structures of the composition expression and the extended composition expression classes. an extended composition expression inherit the same set of fields as of a regular composition expression, plus the composition conditions and assignments.

As mentioned before, the extension point in the base UCA can be based on states or transitions. These states or transitions can be determined in different ways, such as explicit expression of a set of states or transitions, or determining the subset of transitions carrying a certain label. Based on these three ways of expressing the extension points, the composition expression ca be defined in three environments. The user can choose to define the composition expression based on each of these three types of extension points. Figure 16 shows the class structure of the extension point classes which shows the hierarchy of their inheritance and derivation.

The user needs to choose the base and referred UCAs or EUCAs among the current set present in the specification. Furthermore he needs to determine his intended

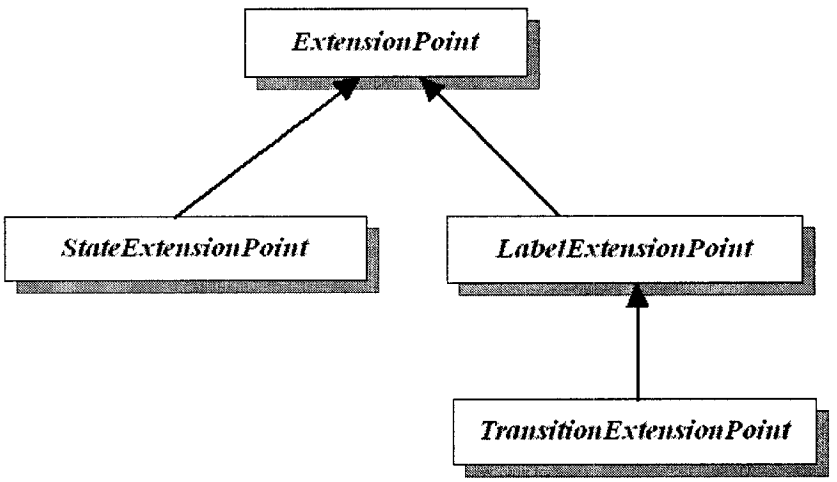

Figure 16: Class hierarchy of extension points

composition operator from the list of previously implemented operators. Then, according to the type of the extension point he has chosen, he has to choose the subset of states or transitions from the base UCA. These settings are provided in easy to use GUI interfaces. An optional check is provided for the designer to choose if he wants the final result of the composition to be minimized or not. This check box is also available in the respective dialog. After determining the composition expression, the new expression is added to the list of expressions in the specification, and is shown in the composition expression frame in the tool. Figure 17 shows an screen-shot of defining a composition expression based on transition as extension point.

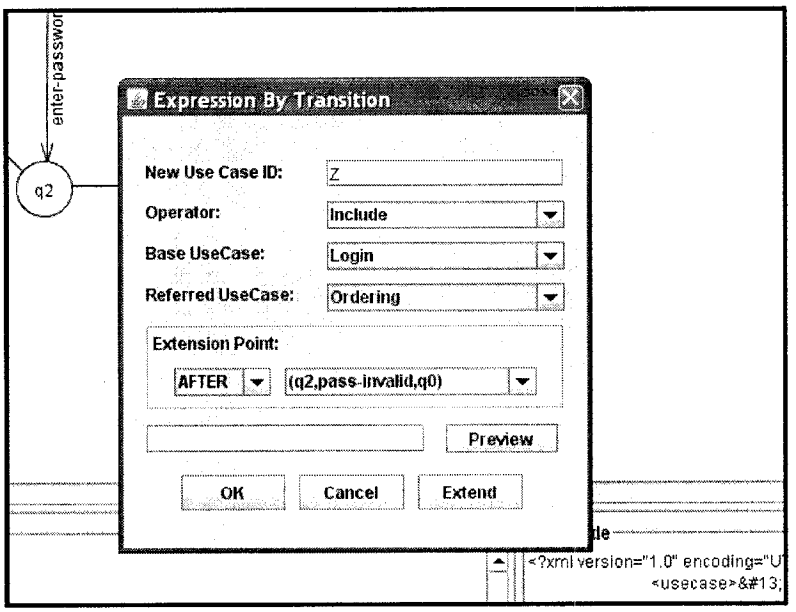

Figure 17: Defining transition based composition expression

6.5 Verification Environment

Aiming at behavioral verification of the generated UCA models, UCOMV provides a verification platform for the checking of UCA model compliance over desired properties. In the specification interface of the tool, the UCA models are interfaced for the verification on an interactive environment. In this environment, a model checking tool is integrated to the tool which is the core of the verification. The UCA and EUCA models, as will be detailed in Chapter 8, are transformed to the input language of the model checker, which in this case is a Promela code. The transformation is done following specific rules which are designed to simulate the UCA and EUCA behavior in the target language code. This code is presented in an editing window, for any possible modification from the designer. Also the environment includes a separate form which has been designed to introduce the temporal properties on the model.

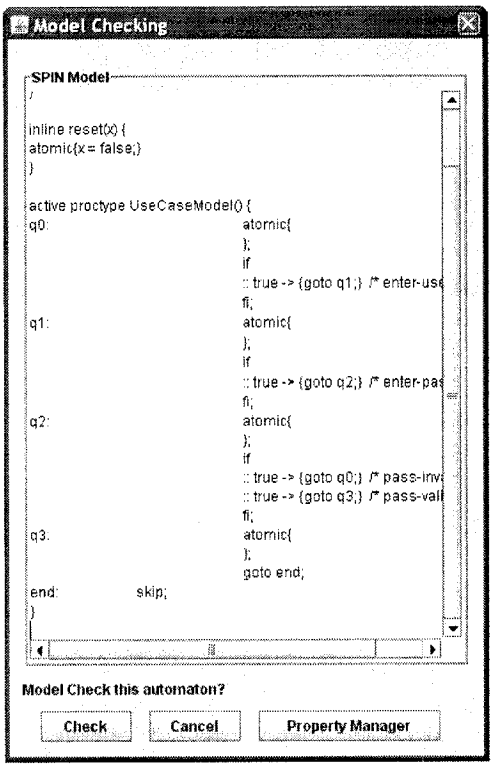

Figure 18: Model checking environment

These properties will be fully explained in the next section, are based on the syntax of LTL temporal properties on the behaviors of the UCAs. It has been noted that the model transformation and the property generation are interfaced in the specification layer. However the core components of the verification module are implemented in the verification layer of the tool. Figure 18 shows an screen-shot of the model verification environment. The code shown in the editor is the Promela code generated for the example UCA.

A separate dialog will help the designer to model the intended behavior. The dialog provides syntactical help to the designer for the expression of his desired property through the first-order logic forming the LTL properties. The designer can introduce

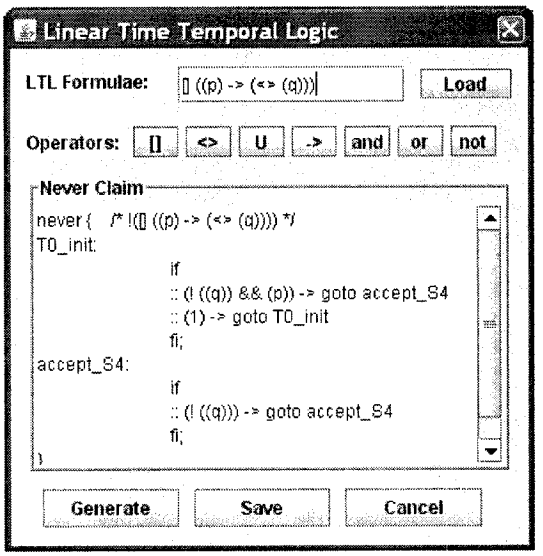

Figure 19: LTL Property Manager

new atomic prepositions and the logical connectors. He can also use some pre-defined property templates provided within the dialog, which include property templates such as Invariance, Response, Precedence and Objective [15]. These pre-defined properties have specific templates seeking specific behaviors from the property, with well-known syntactical structures. They are a part of the model checking environment. Figure 19 illustrates an screen-shot of the LTL property manager and it's property template generator. The property template generator can be initiated using the Load button on the top right corner of the dialog.

6.6 Variable Definitions

The variables which can be used in the EUCA models can have a wide range of semantics. This range starts with primitive variable types, such as **int**, and could be

extended to cover all the programming language semantics and its related structural disciplines, such as arrays, data structures and object modeling. However, the more we enrich variable semantics, the more we add to the complexity of the model and its composition and verification process. Therefore we need to accept the limitations of the implementation of such semantics and define certain constraints. However, we can define a generic platform for the possible extensions to any of these semantics in future.

Variable Types

So far, three basic variable types has been introduced in UCOMV which could be used in the extended UCA models. These three types are of **boolean**, **int** and **float**, which are considered as primitive variable semantics in any programming language and are implemented with their common logical semantics in the Java language. The **boolean** represents a logical boolean values of *true* or *false*, while **int** represents the integer values of the natural numbers. Finally **float** act as the float values of the real numbers in the programming semantics. The variables are encapsulated in the class named **Variable**, which contains a string field as a unique name, the value of the variable in the range, and the model which owns this variable.

The syntax of each condition and assignment could vary to form new values from these primitive types. This could include determining an operation on the value of a variable or assigning the value of this variable to another one. To give an example,

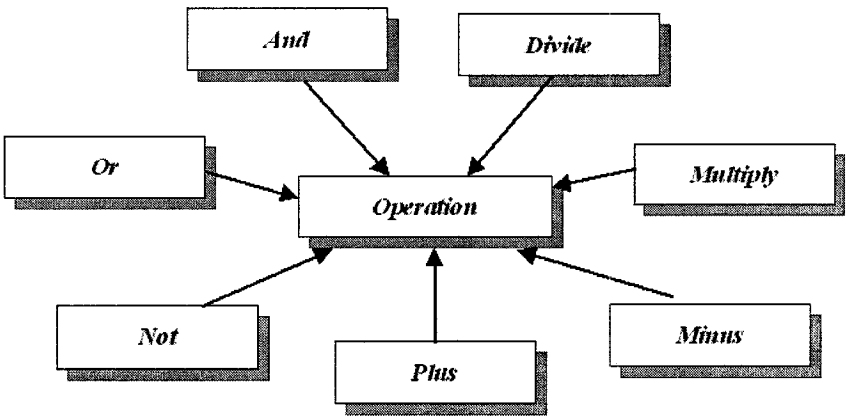

Figure 20: Class structure of the variable operations

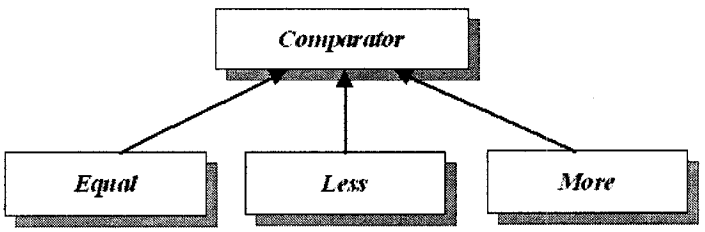

Figure 21: Class Structure of variable comparators

the syntax could be of the form $A = 5$, $A = A + 1$ and $A = B$. According to the type of these three variables, we implemented the logical operations needed to form the whole semantics for the condition and assignment definitions. The operations include *add*, *deduce*, *multiply* and *divide* for the integer and float values and logical *and*, *or* and *negation* for the boolean semantics. The abstract class of **Operation** carries the interface of these operations and should be implemented for each logical semantics in the operators. Figure 20 illustrates the structure of the class derivations for the operations dealing with our basic variables.

We have also defined three types of relative comparators to perform the comparison between the values of two variables of the same type. These comparators are used in the conditions, and would be used to evaluate the validity of the condition during the execution of the EUCA. The three comparator types are *equal*, *more* than and *less* than comparators with their common semantics in the programming languages. Like the operations, they are interfaced in the abstract class of **Comparator** which would be overridden for each of these comparator semantics and any possible future comparators. Figure 21 shows the class structure of the comparators.

The class **Condition** encapsulates a transition pre-condition containing two expressions which would be compared according to their value and type. Also the class **Assignment** carried the needed variables for the variable assignments in the transitions. It contains a variable and expression, the value of which would be assigned to the variable. Figure 22 shows an screen-shot of the environment for defining the guard conditions and assignment for an extended transition.

Local vs. Global Variables

Each variable available to be used in conditions and assignments have an scope. The scope of each variable could be the very EUCA that defines and use it, or the whole specification meaning that that variable is a shared variable among the EUCAs. Therefore the variables available for use in the scope of each EUCA could be the *local* variables for that EUCA or the *global* variables of the specification. Local variables

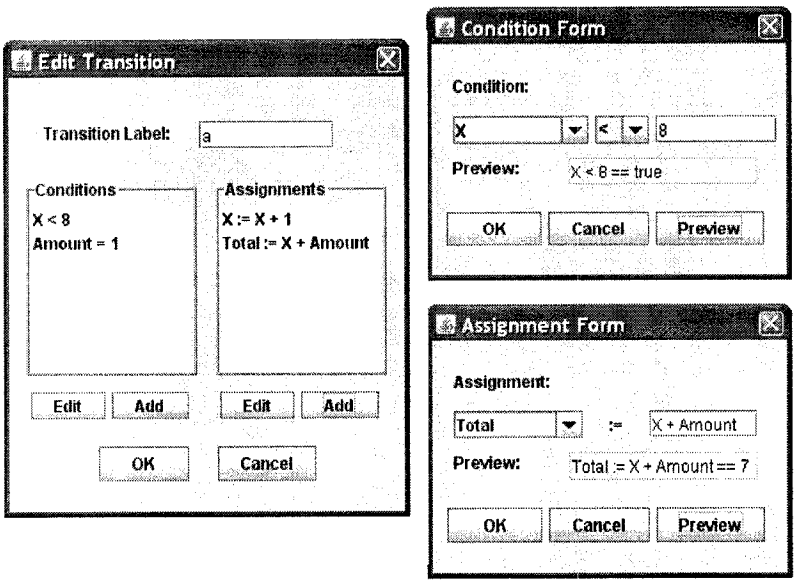

Figure 22: Adding conditions and assignments for an extended transition

act only on the scope of the EUCA that defines them and their values are not available on other EUCAs. Moreover, each specification contains a set of variables which are globally defined for the specification and are common between all the EUCAs. The values of these variables are accessible within each EUCA in the specification and act as shared variable values through the specification and among the EUCAs. Figure 23 shows a list of global and local variables of the example EUCA. In this list, you could see each variable with its definition type, scope (L or G) and the initial values.

6.7 State Coloring

Each designed UCA within the editor is assigned with a color that will be used for the coloring of its states. The type of the color chosen is used to draw the states at the time of design. These colors will further be maintained during the composition process and

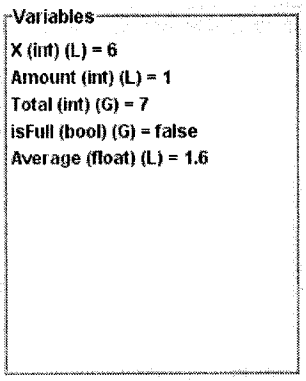

Figure 23: Global and local variable list

newly created states of the new behavior are colored according to the color of their original UCA. Therefore the states of the two composing automata could be traced according to their color. As each state holding the color of its parent, traces of the original UCAs can be tracked in the composed automaton. This coloring technique will help the designer for a better understanding of the composed behavior, and a visual study on the origin of the possible failure in the behavior of the composed UCA. Moreover, as the use cases get more and more complex through several increments, each involving UCA can easily be tracked in the compound UCA which draw an overview of the roots of the newly created requirement and the separation of the concerns within the base use cases. The colors used to characterize the states can be chosen from the list of primary colors, or be customized in the whole range of RGB colors.

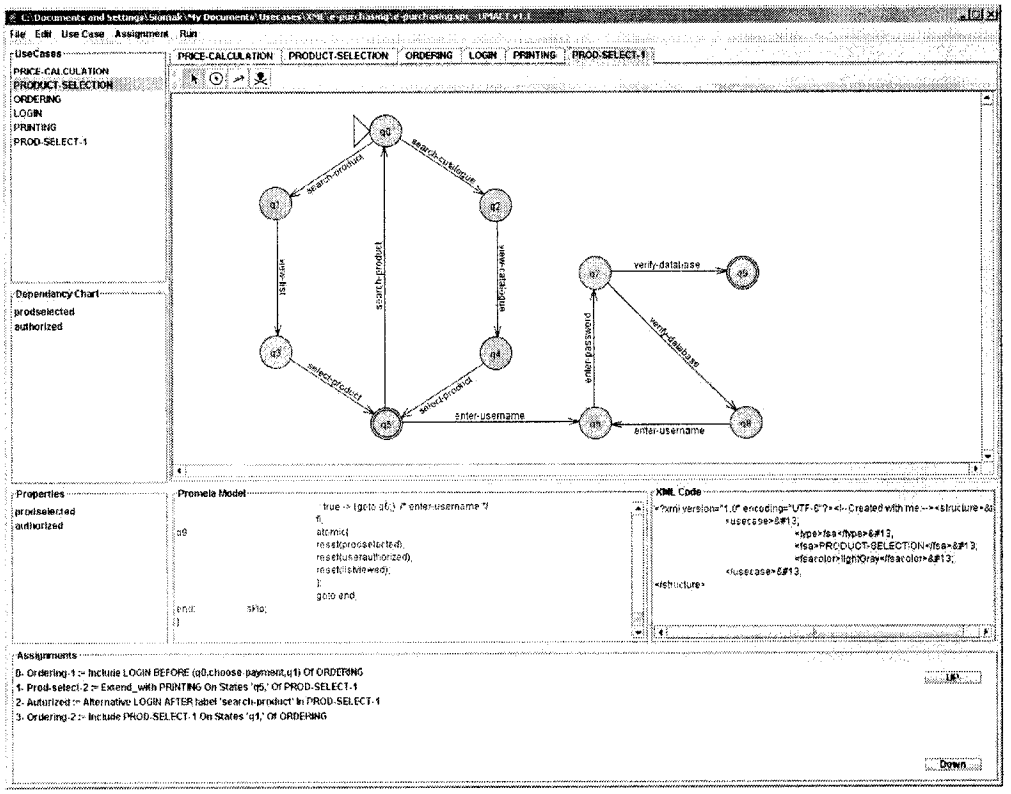

Figure 24: UCOMV

6.8 XML Schemas

A specific XML schema has been designed in UCOMV for the coding of the UCA models and the whole specification which is used for their storage and retrieval as well. Each UCA or EUCA can be singly exported as a proper XML file, which could later be imported to other specifications. This importing and exporting are done using an XML parser with a proper XML schema for defining the structure of the UCA and EUCA. The schema includes elements for retrieving the set of states, transitions, state colorings, and atomic properties. It also retrieves some parameters for graphical presentation of the automata in order to preserve the graphical layout that

the designer has intended for. This information include the location of the states in the screen and ids of the states. These information preserves the latest layout of the automata to be retrieved for the cases of exporting and importing UCAs and EUCAs.

Furthermore, the specification, as the set of UCAs and EUCAs, the list of composition expression, and some other information such as composition history and the hierarchy of the dependencies among the UCAs and EUCAs are stored using the XML schema that we specifically designed in the UCOMV for this purpose. The following code shows the XML code generated for the example UCA shown in Figure 13.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Created with UCOMV -->

<structure>
    <usecase>
        <type>fsa</type>
        <fsa>Login</fsa>
        <fsacolor>yellow</fsacolor>

        <!--The list of states.-->
        <state id="0">
            <x>91.0</x>
            <y>92.0</y>
            <color>yellow</color>
            <initial/>
        </state>

        <state id="1">
            <x>284.0</x>
            <y>92.0</y>
            <color>yellow</color>
        </state>

        <state id="2">
            <x>284.0</x>
            <y>257.0</y>
            <color>yellow</color>
        </state>

        <state id="3">
            <x>479.0</x>
            <y>257.0</y>
            <color>yellow</color>
            <final/>
        </state>

        <!--The list of transitions.-->
        <transition>
            <from>2</from>
```

```
            <to>0</to>
            <read>password-rejected</read>
        </transition>

        <transition>
            <from>2</from>
            <to>3</to>
            <read>password-validated</read>
        </transition>

        <transition>
            <from>0</from>
            <to>1</to>
            <read>enter-username</read>
        </transition>

        <transition>
            <from>1</from>
            <to>2</to>
            <read>enter-password</read>
        </transition>

    </usecase>
</structure>
```

6.9 Conclusion

In this chapter, we have presented the specification layer of the tool in details. We have shown how this layer of the tool provides the necessary data structure and platform for interfacing our notion of specification as the set of UCAs and EUCAs and the list of composition expressions. In the next chapter, we will have the same detail study on the composition engine of the tool and present the core components in the layer implementing the of the composition theory and its implementation necessities.

Chapter 7

Composition Engine

7.1 Introduction

The implementation of the composition theory presented in Chapter 4 is done in the composition engine layer of the UCOMV. This engine provides the implementation for the steps needed in the composition such as builder generation and clone synthesis. It also provides the implementation of the algorithms used for the composition steps, including the label matching algorithm, automata connected component algorithm and transition removal algorithm.

The chapter is organized as follows. Section 7.2 describe how each composition expression are interpreted and the necessary information for the composition are retrieved from it. In Section 7.3, we discuss the process of synthesizing the builders

which are the inputs to the merging algorithm. Section 7.4 discusses the implementation of the merging algorithm which is based on the label matching mechanism. Section 7.5 details the post processes that are needed to be perform on the resulting automaton from the merging algorithm. In Section 7.6 we discuss how the implemented composition approach has been extended for composing the Extended UCAs for their composition conditions and assignments and finally Section 7.7 concludes the chapter.

7.2 Expression Evaluation

Defining a list of composition expressions as an increment is to enhance the specification with new UCAs and EUCAs with more compound behaviors. The new UCA can be designed in the automata design editor, ot be composed and generated from existing ones. The notion of the composition of new UCA is defined through composition expressions and the semantics of the composition operators in them. Each expression defines a new behavior from the existing ones. These expressions are to be evaluated and the new behavior is to be synthesized accordingly. The evaluation of the composition expressions are performed within the composition engine of the tool.

Composition expressions and extended composition expressions are evaluated within a class called ExpressionComposer. An instance of this class is in charge of evaluating the composition expression and composition the new behavior. Different steps

of the composition, including builder generation, merging mechanism, and transition removal algorithm are implemented in these composers.

The expression composer creates the new behavior according to the operator semantics in the expression. The composer produces respective builders for the operators, compose them using the merging mechanism and perform the removal algorithm on the resulting intermediate automaton. It also determines the final states of the UCA according to the specified semantics. Each of these steps along with their implementing components are detailed in the following sections.

7.3 Builder Synthesis

The expression composer component generates the respective builders from the base and referred UCAs. As described in Chapter 4, the semantics of these builders are encapsulated in the composition operators. These semantics are implemented in the operator classes and the expression composer component uses the instantiated operator class in the expression for this purpose. The expression composer generates a clone of the two UCAs and performs the operators' semantics to generate the intended builders.

Base Builder

An abstract implementation of the operators encapsulates the generic semantical tasks performed by all operators. These tasks include the templates for the generation of the base builder with respect to certain semantics, and the generation of the referred builder which is independent from the compositional semantics and shared among all the operators. An abstract class called Operator is in charge of performing these tasks. Two primary methods are implemented in this class which are inherited by the The method named getReferredBuilder implements the process of generating the referred builder while the method named getBaseBuilder is in abstract form and is to be overridden in the derived classes of operators. This generic design of operator enables the tool for future extensions over new compositional semantics and respective operators.

Six operators with unique compositional semantics and their respective base builder templates has been implemented. They are named as Include, Extend, Alternative, Refine, Graft and Interrupt. Figure 25 shows the derivation of these operators from the class Operator. The semantics of these operators for the composition of the behaviors are detailed in Chapter 4 of the thesis, and in [31].

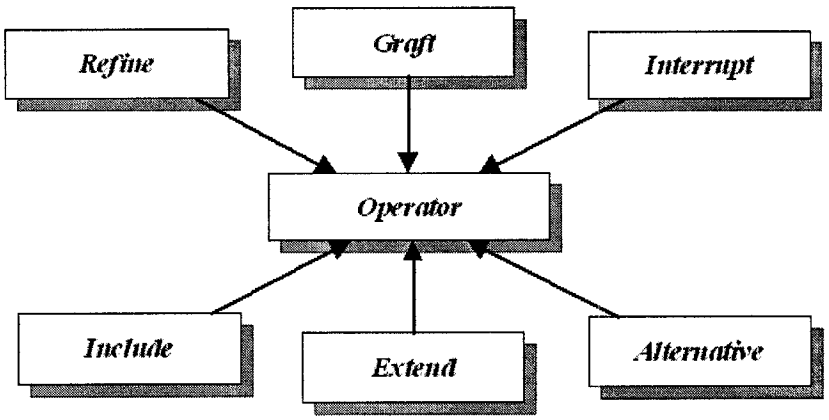

Figure 25: Derivation of Operator classes

Referred Builder

As described before, the referred builder is synthesized independently from the semantic of the composition operator in the expression. In fact, the referred builder is the extension of the original referred UCA with an additional 'begin' state, and two added transitions connecting the final states of the UCA to this new initial state for the builder. These two new transitions are labeled as begin and end. Other transition labels of the automaton are relabeled with adding a postfix showing the index of extension point being merged. This labeling is to guarantee the distinction of the alphabets of the two automata begin merged, and is to be redone after the composition. This semantic is implemented in the abstract Operator class in the method getReferredBuilder, and deployed for all the operators derived from this class. The other methods in the Operator class are abstract interface methods and should be implemented in the derived classes according to the intended compositional semantics

of the operator. As an abstract Java class, the Operator class can not be instantiated directly and objects of its type should only be created through the implementing classes derived from it.

7.4 Merging Algorithm

Each composer uses the *merger* component for the composition of its generated builders. The merger component is in charge of the composition activities and is used to run the composition algorithm. The class called Synchronizer provides a pure implementation of the label matching mechanism which is the basis of the composition algorithm. For the evaluation of each composition expression, the merger component creates an instance of the Synchronizer class with the base and referred UCAs as its parameters. The Synchronizer class provides the implementation of the label matching mechanism for the two given automata in the method called synchronize().

The merging algorithm works on the basis of composing the two automata according to their shared transition labels. This means that the resulting automata from this algorithm would carry a transition with the shared labels of the two automata, and the rest of the transitions of each automata which have separate labels. In our composition approach, these shared labels are only the two added transitions labels of *begin* and *end*. The input to the merging algorithm is the two generated base and referred builders. The intersection of the sets of alphabets of the two automata in

the merger component should just be the added transitions of *begin* and *end* in the builders. This is guaranteed in the composition engine through a renaming procedure on the transition labels done through the cloning process of generating the builders in the composer component. The implementation of this algorithm is deployed for both UCA and EUCA compositions.

Connected Component Algorithm

The resulting automaton of the merging algorithm and the label matching mechanism contains a set of transitions produced as a subset of the Cartesian product of the transition sets of the two base and referred builders. This set of transitions and their respective states are not necessarily representing a connected graph that can be used as a UCA automaton. Many states would be unreachable from the initial states and therefore, transitions firing from and to these states could not be part of any traces and scenarios of the UCA behavior. Therefore, the connected component of the resulting graph of the merging algorithm would be extracted and a clean UCA which is free from redundant states is given. This is done through an specific algorithm we have implemented for this purpose. First a reachability analysis is performed and each single state is determined to be reachable from the initial state. The unreachable states, which are redundant states, are marked. These states are then removed from the automaton and the result would be a connected graph representing our automaton. Each state in the resulting automaton would be a part of at least a scenario in the set of scenarios.

7.5 UCA Post-processing

The result of the merging algorithm is an intermediate automaton containing traces of the base and referred UCAs according to the composition semantic, which needs further treatments to form the new UCA partial behavior. The *UCA processor* component performs post processes on the intermediate automaton. It recovers transition labels which were renamed during the cloning, and also determines the final states of the new UCA according to the semantics of the composition operator.

7.5.1 Final States

The final states of the resulting automaton are determined according to the compositional semantics of the operator. This means that in the case of Include and Extend and their deriving operators, the final states of the resulting automaton would be the states generated from the final states of the base UCA only. However for the Alternative operator and its deriving operator, the set of final states would also contain the states generated from the final states of the referred UCA. This is because of the nature of their composition semantic, and that in the case of Alternative and its children, the referred behavior is interrupting the base behavior without any return to the original one. And therefore the final states of the referred UCAs should also be considered in the final product in order to preserve the scenarios passing to the interrupted section of the automaton. This is all done in a method called determineFinalStates as a part of the class ExpressionComposer.

7.5.2 Removal Algorithm

The UCA processor component also preforms another important process on the resulting automaton from the composition. It removes the added transition of *begin* and *end* which served as the shared labels during the composition. As we discussed before, these transitions are added for generating the builders, and served during the composition as the only common label between the sets of the labels of the two builders. They are to be removed at this point in order to give a clean UCA representing only the traces of the scenarios of the base and referred UCAs.

The base of this algorithm, as discussed in chapter 4 is the classical lambda transition removal of the automata theory. In this regard, the two begin end transitions are treated as null transitions and are removed from the traces of the automaton, without harming the rest of the parts of the traces carrying these labels in them. The original lambda transition removal carries the approach by processing the whole automaton since the null transitions might be present in any number and in any place in the traces. However, in our resulting automaton, we know that we only have two of these transitions, and so we customize the algorithm to be preformed only on these two cases. With this customization, we reduce the complexity of the algorithm to the order of traversing the transition set of the automaton. The algorithm in tool is implemented in the class called TransitionRemover.

7.5.3 UCA Minimization

As an option, the designer could choose to have the resulting UCA from the composition as a minimal automaton. This option was presented at the specification layer and is a part of the description of the expression. If the minimal option is checked, an automaton minimization algorithm is performed on the resulting UCA from the removal algorithm. The algorithm which is implemented in UCOMV is merely the classical minimization algorithm from automata theory which applies the state grouping technique to find the states which could be merged while preserving the same set of traces.

The algorithm starts with transforming the given UCA to a deterministic finite state automaton. The UCAs structure can have lambda transitions and multiple same labeled transitions from a states which could result in non-deterministic structures. Therefore this transformation is needed as the initial step before the original minimization algorithm begins. After retrieving the deterministic version of the UCA, the minimization algorithm starts with the initial state, holding initially all the states of the automaton. These states are gradually expanded to separate states in each iteration of the algorithm, and hence the ones which could possibly be merged are eventually held unexpanded. The states which represent more than one merged states from the original UCA in the minimized automaton are colored as gray and are distinguishable from other states not merged with any other one. Other non-merged states are colored as the color of their original states in the UCA. The output of

the algorithm would be the minimized UCA, containing the same trace of scenarios expected from the composition with equal or less number of states than the resulting UCA from the removal algorithm. It is notable that this option enables the designer to have the final UCA as such that the structure of the referred UCA is not unfolded inside the base UCA, which happens on the removal algorithm. This would be the last phase and the product of the UCA Processing component is the final new UCA model and is to be added to the existing set of UCAs.

7.6 Composition of Extended UCAs

The composition of the extended UCA models are performed with the extension of the same approach used for the UCAs. The new behaviors are composed from the evaluation of the extended composition expressions, and will pass the same steps of the UCA composition. First the two base and referred builders are generated according to the compositional semantics in the operator. Then these builders are used as the inputs to the merging algorithm based on the label matching mechanism. It is notable that the decoration of the UCA model with variables would not affect the merging algorithm, and the label matching mechanism would be performed for the composition of UCA models. After the merging, the intermediate EUCA would be refined according to the extended version of the removal algorithm, which would be described in the following section.

Extended Builders

The generation of the builders in the evaluation of the extended composition expressions follows the same synthesis rules of the UCAs, except that the two transitions of the *begin* and *end* would carry the composition conditions and assignments expressed in the extended expression. The added *begin* transition in the base builder would be decorated with the composition condition in the expression as the pre-condition and the set of input variable assignments as the assignment set of its transition. Also the added *end* transition in the base builder would be decorated with the set of output variable assignments in the extended expression as the transition assignment set.

It is notable that the variables used in these three sets of conditions and assignments should be limited to the valid variables in the scope of the base EUCA. This include the local variables of the base EUCA, as well as the global variables defined in the specification. This limitation is due to the fact that some variables in the referred EUCA may not be initialized before the execution of its behavior, and hence they may not be evaluated in any condition or assignment expressions. The referred builder is synthesized following the exact approach in the evaluation of the UCA composition expressions. Furthermore, the composition is performed on these builders through the same merging algorithm and following the same label matching mechanism for the regular UCA composition.

Extended Removal Algorithm

The removal algorithm used for the removal of the *begin* and *end* transitions would be an extension of the removal algorithm used in the UCA composition [31]. In the intermediate automaton from the merging algorithm, the two *begin* and *end* transitions would carry the set of composition conditions and the input and output variable assignments. Therefore, as the first step and before the original removal algorithm is implemented, these conditions and assignments should be propagated to other transitions in the new EUCA. The pre-conditions of the *begin* transition would be propagated forwardly to all the outgoing transitions from the incoming state of the *begin* transition. In contrast, the propagation of the variable assignment set is done backwardly on the incoming transitions of the outgoing state of the *begin* transition. The similar approach would be applied for the removal of the *end* transition [31].

After this phase, the intermediate extended automaton is ready for the original transition removal mechanism, and the two transitions would be removed according to the algorithm presented in chapter 4. These two phases are implemented in the extended class of *ExtendedTransitionRemoval*, which is a derivation from the class *TransitionRemover* implementing the original removal algorithm for the UCA composition.

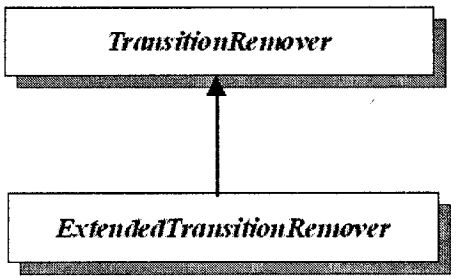

Figure 26: Derivation of transition remover classes

7.7 Conclusion

In this chapter, we discussed in details, the composition engine of the tool which provides the necessary data structures and implements the proper algorithms for deploying the composition approach presented in Chapter 4. This layer of the tool gets the description of composition of a new behavior in the composition expressions as its inputs, and delivers the composed and processed UCA or EUCA expected from the composition expression, hence a distinct layer with interfaces with the specification layer and the verification layer. In the next chapter, we will discuss the verification layer of the tool and show how the integration of our model to a model checking model has been done.

Chapter 8

Model Verification

8.1 Introduction

As the requirements specification is gradually being formed in each increment, the newly generated models are subjected to check for breaches from the desired behavioral properties. Through the composition of two UCAs, the designer seeks to acquire a specific series of behaviors with the composed UCA model. But the semantic of the composition used as operator for the composition might not produce the exact behavior that the designer targets for. These breaches can be detected using a verification mechanism designed in UCOMV, and the violating traces can be found and used to debug for the root of the problem. UCOMV provides an integrated model verification platform for helping the designer to achieve this goal. In this chapter we will discuss the details of model verification layer of UCOMV and show how it could be used as an assistant to produce proper desired specifications in our approach.

The chapter is organized as follows. Section 8.2 discuss how we aim at deploying a model checking mechanism on our scenario based approach of generating the system specification. In Section 8.3, we present the tool and platform that we have chosen to be integrated in UCOMV for the verification purpose. Further in Section 8.4, we show how we transform our formal UCA model to a proper code for the model checker to represent the behavior of the UCA, and show how this parsing approach has been implemented in UCOMV to be expendable to other model checkers to the suite. We also discuss how we restrict the formal model used in the target model checker to our UCA. Section 8.5 shows the steps implemented to perform the model checking in UCOMV, and how the results of the verification process is used to extract the error traces generating the failure scenarios. Finally in Section 8.6 we note a limitation of our study on the inheritance of the behavior properties on the composed models and show the direction for this study in future. Section 8.7 concludes the discussion on model verification in UCOMV.

8.2 Behavior Verification

Each UCA represents a set of scenarios of behaviors from the system. As a designer, the UCOMV user is supposed to produce the proper set of behavior from the specification generation process supported in UCOMV. This set of behaviors are to be produced by designing new UCA models and use them to compose new compound

models according to compositional semantics. However, its always challenging to foresee the proper UCA model with the exact desired set of behaviors, both from the designing process and composition using compositional semantics of the operators. Therefore the designer might want to check for the properness of its product and verify them over its target set of behaviors. This is done through defining and verifying proper liveness and safety properties on the set of scenarios.

UCOMV provides a platform for defining a set of safety and liveness properties for each UCA model. These properties are verified over the behaviors of the UCA and possible breaches can be followed as traces of the UCA. The set of properties include the checking of the model over desired and undesired behaviors in the composed UCA. Hence the model can be checked for the presence or absence of a specific set of behaviors and respective properties representing them. It is notable that the steps of the composition mechanism are guaranteed to be free of implied scenarios, and the verification at this layer aims at helping the analyst to verify his desired set of behaviors in the generated UCA model. The SPIN model checker [22] is integrated into our framework for this purpose, and *Linear Temporal Logic (LTL)* formalism is deployed to specify the temporal properties on the model's behavior.

8.3 Promela and SPIN

In the theory of automated verification, to verify a prospective system, two concepts should be defined: the set of facts we want to verify, and the relevant aspects of the system that are needed to verify those facts, such as a model representing the system. Furthermore, two distinct approach for this verification has been developed in the literature. In the first approach, a mathematical proof steps are provided for a specific theorem based on basic models. However in the second approach, an exhaustive check over the behaviors of the prospective system is performed, and possible violations are extracted. Generally the first approach is referred to as the Automated Theorem Proving, while second is called the Model Checking. In the model checking approach, a behavior specification describes what is possible in our model, while a property specification defines what are we checking for. In our approach, our verifiable model is the generated UCA model, and the facts to be verified are the behavioral properties in terms of LTL expressions. We use the model checking technique to verify the model conformance over the desired properties, and use the SPIN model checking tool for this purpose.

The **SPIN** (**S**imple **P**romela **IN**terpreter) is a popular tool for analyzing the logical consistency of concurrent systems [22] which is originally developed by Gerard Holzmann in 1991. Formal concurrent systems in SPIN are described in a modeling language called *Promela*. Promela is a specification language to model finite-state

systems which is used to specify the behavior specification. It has a close syntax to C language, but it is just a modeling language rather than a whole programming language semantic. Furthermore property specifications are generated from the given LTL properties. SPIN takes the Promela specification for the behavior model and for the properties, generates the property specification from the LTL property, and automata from the Promela code. Then it uses the automata to check for the language emptiness of these two models. In UCOMV, the behavior model is generated from the UCA model produced in the specification layer. The property model is generated in a separate environment with the help of SPIN, and is added to the behavior model as a never claim.

8.4 Model Transformation

As the first step, as the designer chooses to enter in the model verification environment, the given UCA in the specification layer is transformed to a proper Promela model for SPIN within the *UCA-Promela Transformer* component. This component provides an abstract template of parsing rules for the UCA model to generate the proper code for the target language of verification. These rules specify how the UCA model is translated to the target modeling language and how to generate codes simulating the automata behavior of the UCA in the language of the verification tool. It determines how the states are specified in the code, and how transitions from the automata are coded between these states. In the current version of the tool, the

transition rules from the UCA model to the SPIN Promela language of the SPIN model checker have been implemented. In the transformation of a UCA model to the Promela code, state labels are specified as the labels of code lines. Each of these labels define a block of code which determines the actions happening from that state. The block is labeled with the state label and is referred to with this label for the modeling of transitions to this state. Furthermore the atomic property predicates which are valid in that state are coded inside the block. The following code shows an example coded for state q1 in the UCA given in Figure 27.

```
accept_q1: atomic {
             set(a);
             reset(b);
             reset(c);
           };
           if
           :: true -> {goto q2;}  /* b */
           :: true -> {goto q0;}  /* d */
           fi;
```

Atomic properties are those which are introduced in the specification layer on the UCA, and are used further to specify desired properties to be checked. a, b and c are the examples of such property predicates which the code shows a is valid in this state, and b and c are not. As for the transitions fired from the state, the set of transitions are modeled with a set of if statements, each representing the transition firing in the case of the transition label being seen and Promela code being traversed by the SPIN.

In the Promela code, to form the notion of final state, we add another extra state block called end state, which represents an additional state used to redirect all the transitions targeting a final state to this state. This state is the last state coded in

the Promela code, and contains no transitions fired from it. It basically act as an empty accepting state, directing the Promela script to end the execution of the code. Furthermore, we add an empty transition to this state from all the states which are marked as final states in the UCA. Therefore all the final states of the UCA model would have an additional empty transition to this state. This approach would emulate the notion of final state of our UCA model in the Promela code. As you can see in the given example code, state q2 is a final state and has a extra coded transition to the additional state block of end at the end of the code. The notion of the initial state is emulated with having the UCA's initial state coded as the first state block in the Promela code and hence, the first block of code to be traversed by the SPIN at the time of execution. When the model checker first starts to traverse and execute the code, it starts from the beginning with this initial block of state which brings our intended notion of initial state in the UCA.

Moreover on the UCA, we have also the notion of 'accepting' states, which are the states considered as accepting states in SPIN. On the specification layer, the designer has the option on each state to declare that state as the accepting state of the automata. This would mark that state as the relevant notion of accepting state in Promela. These states are respected with their marking label of the block of code which their labels would start with addition of "*accept_*". We note that accepting states are the states used by SPIN to prove either the absence or presence of infinite runs that traverse at least one accept state in the global system state space infinitely

often. These states and their mechanism are used to prove LTL liveness properties, as an example, and are different than the notion of final state which mark the termination of the execution of the automaton.

These coding conventions help us to emulate the UCA behavior as a Promela code for SPIN. They are abstracted and could be valid for any modeling language for any other model checker. Therefore the rules can easily be overridden for generating codes of target languages used in other model checkers to the suite, hence enabling the tool to plug the interface to other model verification tools. The extension of the verification layer to support the UPPAAL model checker is seen as part of our future extensions. Figure 27 shows an example of a simple UCA model and its relatively generated Promela code. The parsing rules for the Promela has been implemented in the class called **AutomataParser**. This class implements the abstract method called parse() for transforming the given UCA model to a proper Promela model, and can be overridden in future derived classes of parsing for other model checkers. This method assumes the parsing conventions we discussed for generating proper codes of Promela modeling language from the UCA automata model.

As described in Chapter 5, behavioral properties in UCOMV are defined using the LTL temporal logic formalism which is the base for the SPIN model checker. The class called **LTLProperty** encapsulates the specification of an LTL property formalism. This class includes the syntax of a property written respecting the LTL logic.

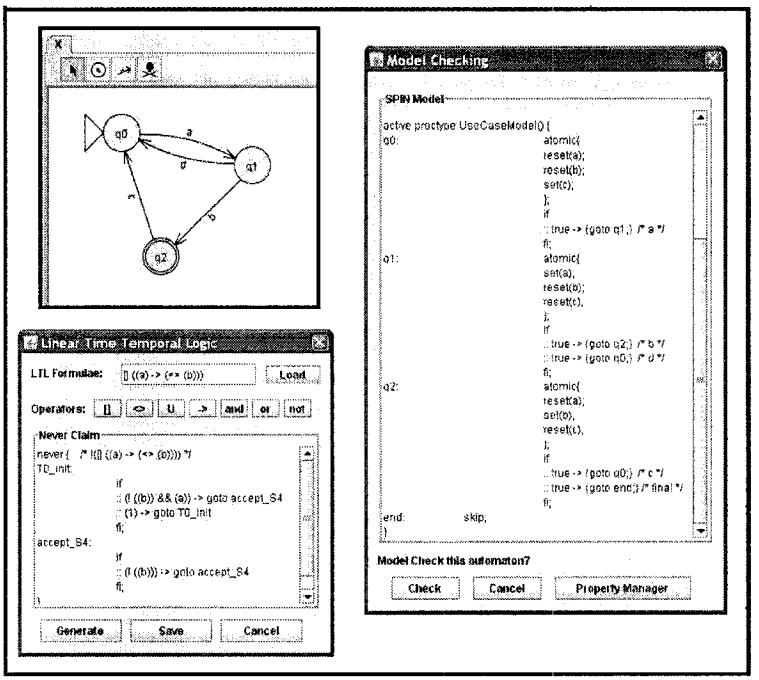

Figure 27: Example of an UCA, and its Promela models in UCOMV

It also includes the never claim which reflects the given property transformed to a proper Promela model. This whole is done in the *LTL Property Manager* component. This claim along with the behavior model from the automata interpreter provides a complete verification model as the input to the SPIN, and is used to check for correctness of the property on the UCA model. Following is an example of a Promela code generated in UCOMV to represent the simple UCA model and a simple liveness property. It emulates a UCA with three states and three transitions and two atomic properties of p and q. You can see how these properties are set or unset in the states, and how the LTL response property on p and q is transformed to the never claim in the code.

```
bool p = false;
bool q = false;

inline set(x) {
   atomic{x = true;}
}

inline reset(x) {
   atomic{x = false;}
}


active proctype UseCaseModel() {

q0: atomic{
      reset(p);
      reset(q);
    };
    if
    :: true -> {goto q1;}  /* a */
    fi;

accept_q1: atomic{
      set(p);
      reset(q);
    };
    if
    :: true -> {goto q0;}  /* d */
    :: true -> {goto q2;}  /* b */
    fi;

q2: atomic{
      reset(p);
      set(q);
    };
    if
    :: true -> {goto q0;}  /* c */
    :: true -> {goto end;} /* final */
    fi;

end: skip;
}


never {  /* !([] ((p) -> (<> (q)))) */
    T0_init:
    if
    :: (! ((q)) && (p)) -> goto accept_S4
    :: (1) -> goto T0_init
    fi;

    accept_S4:
    if
    :: (! ((q))) -> goto accept_S4
    fi;
}
```

8.4.1 UCA as Promela Model

In order to enable the user with the capabilities of verifying the model with LTL properties, we add some notions to the original UCA definition. In fact, the property

predicates that the designer define on states, and the notion of accepting states are the two concepts that we need the designer to define in order to enable him defining LTL properties. These two notions are not present in the theory part of the UCA, and are not parts of the original formal model presented in [32, 33, 24]. These two definition not needed for behavioral modeling using UCA, and are just used to enrich the transformed model of UCA to Promela. We note that the transformed code of our UCA model in SPIN are the restricted model of Promela to represent the UCA and its behavior. In fact the semantic of the behaviors that the UCA is representing is a far less expressive than the model that SPIN is using for its verification. SPIN uses a mathematical model called *Büchi automaton* for the check of emptiness of the intersection of the model from the behavior and the property. The Büchi automaton is the extension of the finite state automaton on infinite inputs and loops. It can be used to detect systems with indefinite final states with infinite number of occurrences of loops. Therefore by definition, the model SPIN uses for his check is far more expressive than our UCA model which could model and check other types of properties and behaviors.

In UCOMV, we restrict the input model to SPIN to a proper representation of our UCA. For this purpose, we use templates for transforming the UCA structure to a Promela code, and simulate its behavior there. This template restricts the model supported by the Promela language to the representing UCA, and use the SPIN to actually run this restricted model and check for its emptiness along the model from the property. This is not a limitation to our approach since in our formalism we don't

aim at modeling that level of expressiveness that Promela is supporting. We aim at at deploying the SPIN and its model to check for the compliance of our UCAs over the properties, at the same level of expressiveness and formality that they are defined in UCOMV. And we add the two notion of state property predicates and accepting states to provide the necessary framework for LTL property definitions.

8.5 Verification and Error Trace Parsing

After transforming the UCA model and the LTL property to proper Promela codes, these codes are given as an input to the SPIN model checker. The verification is done in three steps which are the main steps for the verification using SPIN. These steps are implemented in UCOMV in the class called SPINVerifier. First the SPIN is triggered and the generated Promela code for the UCA and the property are given as the input to the engine. Then SPIN generates a C file, which contains a simulation code for the checking of the emptiness of the intersection of the main Promela code and the code for the property. This C code file needs to be compiled using the 'gcc' compiler, which results in an '.exe' file. This '.exe' file performs the verification for the checking of the model, and returns a report file. The report file contain useful information on the result of the verification, and should be interpreted to check if the model has satisfied the given property. It also helps to produce the possible error trail of the model which violated the property. The interpretation of the report file from the SPIN is done in the class called ReportAnalyzer. This class is used within

the *debugger* component of the verification module in the tool, and determines if the model has satisfied the property. If not, runs the SPIN in the debug mode and use the output of the debug mode to extract the error trail section from the report file, and instantiate the class ErrorTrailExtractor with the generated error trail code. From the given simple example LTL property given in Figure 27, the following report is produced as the original output of the SPIN verification.

```
(Spin Version 4.2.7 -- 23 June 2006)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim         +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 16 byte, depth reached 29, errors: 1
      14 states, stored (18 visited)
       4 states, matched
      22 transitions (= visited+matched)
      12 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.000  equivalent memory usage for states (stored*(State-vector + overhead))
0.299  actual memory usage for states (unsuccessful compression: 88979.76%)
State-vector as stored = 21347 byte + 8 byte overhead
2.097  memory used for hash table (-w19)
0.320  memory used for DFS stack (-m10000)
0.094  memory lost to fragmentation
2.622  total actual memory usage

unreached in proctype UseCaseModel
  line 11, "pan.___", state 6, "b = 0"
  line 7, "pan.___", state 9, "c = 1"
  line 20, "pan.___", state 13, "goto q1"
  line 19, "pan.___", state 14, "(1)"
  line 11, "pan.___", state 21, "b = 0"
  line 22, "pan.___", state 25, "a = 1"
  line 41, "pan.___", state 53, "-end-"
  (7 of 53 states)
unreached in proctype :never:
  line 52, "pan.___", state 11, "-end-"
  (1 of 11 states)
```

From the report you could see that there is an error happening while verifying the property from the line errors: 1. Therefore SPIN needs to be run in the debug mode and the error trail information should be extracted. The SPIN generate the possible

error trail on the reachability graph of the Promela code. This graph is originally generated by the SPIN from the input Promela code in order to traverse and execute the code. The error trail given on this graph needs to be interpreted back as a trace of the UCA model. In output report from the verification, SPIN specifies information such as if the property has been verified and that which lines of the Promela code has not been reached during the execution. If the property is not verified, then the SPIN specifies the cycle it found in the execution of code preventing the graph to reach its final states, and the values of the variables in the states failing the property. According to these information, we can trace which line of the Promela code has been executed in the error scenario, and from the code line, we can determine which code block this line belongs to and hence, determining the UCA state on which this event has happened. Following is the result of the debug running of SPIN on our example for the LTL property.

```
Starting UseCaseModel with pid 0
Starting :never: with pid 1
Never claim moves to line 46 [(1)]
  2: proc  0 (UseCaseModel) line  10 "pan.___" (state 1) [a = 0]
  3: proc  0 (UseCaseModel) line  10 "pan.___" (state 4) [b = 0]
  4: proc  0 (UseCaseModel) line   6 "pan.___" (state 7) [c = 1]
  6: proc  0 (UseCaseModel) line  20 "pan.___" (state 11) [(1)]
  8: proc  0 (UseCaseModel) line  20 "pan.___" (state 12) [goto q1]
 10: proc  0 (UseCaseModel) line   6 "pan.___" (state 16) [a = 1]
 11: proc  0 (UseCaseModel) line  10 "pan.___" (state 19) [b = 0]
 12: proc  0 (UseCaseModel) line  10 "pan.___" (state 22) [c = 0]
Never claim moves to line 45 [(!(b)&&a))]
 14: proc  0 (UseCaseModel) line  29 "pan.___" (state 29) [(1)]
  <<<<<START OF CYCLE>>>>>
Never claim moves to line 50 [(!(b))]
 16: proc  0 (UseCaseModel) line  29 "pan.___" (state 30) [goto q0]
 18: proc  0 (UseCaseModel) line  10 "pan.___" (state 1) [a = 0]
 19: proc  0 (UseCaseModel) line  10 "pan.___" (state 4) [b = 0]
 20: proc  0 (UseCaseModel) line   6 "pan.___" (state 7) [c = 1]
 22: proc  0 (UseCaseModel) line  20 "pan.___" (state 11) [(1)]
 24: proc  0 (UseCaseModel) line  20 "pan.___" (state 12) [goto q1]
 26: proc  0 (UseCaseModel) line   6 "pan.___" (state 16) [a = 1]
 27: proc  0 (UseCaseModel) line  10 "pan.___" (state 19) [b = 0]
```

```
 28: proc  0 (UseCaseModel) line  10 "pan.___" (state 22) [c = 0]
 30: proc  0 (UseCaseModel) line  29 "pan.___" (state 29) [(1)]
spin: trail ends after 30 steps
#processes: 1
a = 1
b = 0
c = 0
 30: proc  0 (UseCaseModel) line  29 "pan.___" (state 31)
 30: proc  - (:never:) line  49 "pan.___" (state 9)
1 processes created
```

You can see the lines of the Promela code involved in each line of the report. From the start of the cycle, we determine the start of the error trail and follow up the transitions of the trail passing through the code. For example the first line shows that line 29 on the Promela code has been executed and the following three lines are the variable evaluations which follow that code. Line 29 on the Promela code belongs to the block of the code related to the state q1. Therefore we determine that the cycle starts at state q1 of our UCA. From the rest of that line in the error report, we see the next command which has been executed is the command goto q0 on the line 29 of the code. This shows us that we have been in state q1 and the next transition which is run by the SPIN had leaded it to the block code of the state q0. From the line 29 of the Promela code, we determine the label of this transition, as d, and reproduce the fact that the cycle in the error trail has been started with firing transition (q1,d,q0) of the UCA. The tool follows the same path for interpreting the other transitions of the error trail report, and extract other transitions fired in the automata which resulted in the violation of the property on the UCA. The tool further sequences these transitions starting with the transition fired from the initial state, and reproduce the UCA trace violating the desired property. All these are implemented in the ErrorTrailExtractor class within the tool. The product of this

process is the violating UCA trace and is then visualized on the respective UCA as traces of the UCA. The analyst can visually track the failure of the property and modify the UCA according to his desired behavior.

8.6 Property Inheritance

When two UCAs compose together and produce a new compound UCA, the atomic properties of the base and referred UCAs are inherited in the composed UCA and would be available for new property verifications on the new UCA. However, for the verified LTL properties on the both parent UCAs, the results are not straightforward in the compound model. Because of the nature of the composition and the base and referred UCA approach, the LTL properties verified on the referred UCA would still be valid for the respective section of the compound UCA. However for the base UCA, there is no proof for the whole compound model to maintain the properties previously verified on the base UCA parent. The validity of those properties and their relationship to possible valid properties to be hold on the new composed UCA needs further detailed investigation and is farther than the scope of this thesis.

8.7 Conclusion

The incremental approach we present in this thesis can help the designer for the verification of the large composed models. This incremental approach can help the analyst to break the complexity of verifying the behaviors on large complex models

into simpler property verifications in each step. In this chapter we have shown how we deploy a model verification mechanism to assist the designer in verification of the model compliance to his desired properties. We detailed on how we transform our model to a proper model in the target model checking tool, and how we parse back the result from the verification and translate it to a behavior in the UCA model. We also have discussed the notions that we add to the original UCA model in the target language, in order to make this verification possible. In the next chapter, we would discuss the interesting feature of traceability in UCOMV and the details of how the feature is designed.

Chapter 9

Traceability

9.1 Introduction

Requirements are subjected to constant changes. The ability to automate the reflection of these changes on the specification is an essentially useful feature for the maintenance of the requirement specifications. These changes may also root in the modifications needed on the composed models after the study of by the designer. As the two behaviors are composed and a whole new behavior is produced, tracking the generating behaviors in the compound model can help the study of the new model. In this chapter, we present an essentially useful feature in the tool which enables the designer to perform this reflection. We will discuss the details of how the feature is designed and implemented in the tool, and how proper data structures are defined in order to support efficient reflection on the UCAs.

The chapter is organized as follows. Section 9.2 discuss two types of UCAs and their difference in terms of their relation to other UCAs. Section 9.3 defines the recursive forward and backward dependencies of UCAs and shows how these dependencies would recursively form a hierarchy of composition history. Section 9.4 defines the idea of forward propagation of structural changes on the dependent UCAs. Section 9.5 discuss the notion of efficient propagation and presents the algorithm of propagation on the hierarchy. It discusses how the idea of defining and using a compositional order on the hierarchy would enables us to reach the minimum number of propagation. Finally Section 9.6 talks about a problem during the propagation and presents its solution that we have deployed. Section 9.7 concludes the chapter.

9.2 UCA Relation Types

Once a UCA is composed and added to the existing set of UCAs, it becomes a part of the specification as a new partial system behavior. The new compound behavior would not constrain the behavior of its composers. However its structure would be dependent to its composing parents. This means that the structure of the UCA which is composed from other UCAs in the specification is not an independent structure, but a product of the structure of the base and referred UCAs.

Accordingly, the set of UCAs in the specification can be categorized to two classes according to the type of their relations to other UCAs. The first category would be

the UCAs which are the product of composition of other UCAs, while the second category are those UCAs which are solely a product of the automata design interface. The definition of the structure of the automata and the nature of their behaviors in the two groups are the same. However, the structure of UCAs in the first group is dependent on their precedents while in the second group, the structure is independent from the other behaviors. We note the first group as *dependent* UCAs and the second group as *atomic* UCAs.

9.3 Hierarchy of Dependency

Different increments in the specification create a hierarchy of composed UCAs which are dependent on each other. Each UCA in the specification may be composed of two UCAs, each of them in turn may be composed from other UCAs. We call this hierarchy as the *backward* dependency of the UCAs. Moreover, each UCA may be used to compose arbitrary number of new UCAs, each of which may be used for new UCAs in other increments of system specification. This hierarchy is called *forward* dependency. The forward and backward dependency of UCAs generates a hierarchy of composition which we call as *hierarchical dependency chart.* The hierarchical dependency chart is formed in the first increment of composition, and is updated during each increment. After the evaluation of each composition expression, the new UCA is inserted into the hierarchy, and the hierarchy is updated regarding the forward and backward dependencies of the newly inserted UCA and the affecting UCAs in the

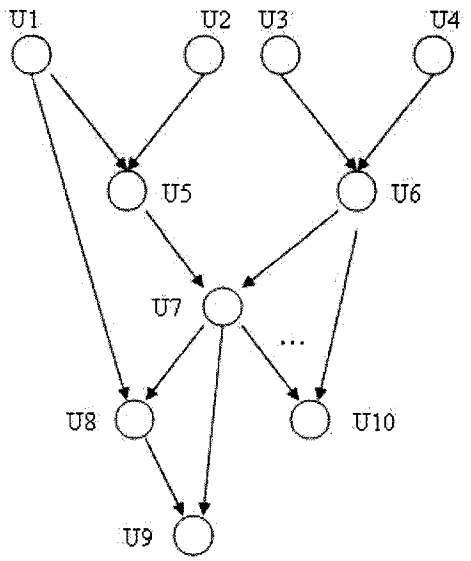

Figure 28: Hierarchical Dependency Chart

hierarchy. Figure 28 shows an example of such hierarchy.

9.4 Forward Propagation

While structurally the same, atomic and dependent UCAs are different according to their compositional history, hence further changes in the specification has different impacts on them. The structure, and therefore the semantics of the atomic UCA can only be changed through the automata design interface inclusively. However the behavior and structure of the dependent UCA can be updated by semantical changes in the affecting UCAs anywhere in the hierarchy. This means that any change in the structure of a UCA through the automata design interface can affect the behavior of all the forward depending UCAs as well. This feature is called *forward propagation* of change in the specification. Forward propagation is presented as an optional feature

in the tool, which means that the analyst can choose to propagate the change, or to destroy the dependency relationship of the changed UCA and have it as an atomic behavior.

The nature of forward and backward dependency is complex. This stems from the fact that each dependent UCA is backward dependent to two involving UCAs directly, and recursively to a sequence of UCAs. On the other hand, it can be dependent forward to a set of arbitrary number of UCAs directly, and a sequence of such sets recursively. This notion leaves us with a complex hierarchical dependency chart along the existing UCAs. Figure 28 delineates an example of such structure. As an example, the UCA U5 in this figure is backward dependent to U1 and U2, and forward dependent to U7, U8, U9 and U10. Therefore any change in the U1 and U2 would optionally be reflected to U5, and in turn recursively to the other UCAs from the forward dependency set of U5.

Forward propagation of changes is done in the *dependency propagator* component of the composition engine as the primary step in interpreting each increment. First, each change in the previously composed UCAs are monitored and marked. These changes include any structural change which affects the behavioral semantics of the UCA, including the set of states, transitions, the initial and the final states of the UCA. As the changes are detected, the UCA is marked and the sequence of forward dependent UCAs affected from this change are determined recursively according to the hierarchical dependency chart. The structural change is then forwardly propagated

in the specification through a minimal recomposition process of the dependent UCAs in the hierarchy.

9.5 Efficient Propagation

During the forward propagation, any repetitive recompositions of UCAs may lead to incorrect propagations and should be avoided. However because of the complex nature of the hierarchy, managing the dependencies would be hard. For example in Figure 28, if there happens such change in U1, the direct following affected UCAs would be U5 and U8. However this change would affect U8 in another way in the hierarchy. The propagated change in U5 would form a new hierarchy of forward propagation which will affect U7, and recursively U8 in the next layer. Therefore the original change in U1 would be propagated to U8 from two paths in the dependency chart. This cause a repetitive propagation on U8 which will result in incorrect behaviors, and inconsistency in the specification. To avoid this problem, the propagation should first be reflected on U5 and hence, U7, before reaching the U8 use case. In a general case, this would lead to the fact that in order to minimize the propagation of changes and the recompositions, for each UCA to be affected, the two possible paths leading to the UCA should be synchronized at the previous level right before the UCA in the hierarchy.

9.5.1 Compositional Order

The solution to this problem is to find a sort of order for the propagation process in the UCAs. Such order would help to prevent the repetitive propagations. The hint for such order lies in the original set of UCAs in the specification, where the dependent UCAs are added. This hint can be used to form a set of orders in each UCA which preserves the order of insertion of the UCA in the specification and can be used to recursively surf the dependency chart to find the UCAs added to the specification "before" and "after" each UCA. Each UCA would have an ordered set of dependent UCAs, and the hierarchy of these orders is detained through the hierarchical dependency chart. This compositional order would be used during the propagation process, and would prevent us from any repetitive propagation while guarantees the forward reflection of changes for all sequences of the affected UCAs, recursively.

The partial order on the set of UCAs would help us determine their privilege on the process of propagation. The order counts such that each UCA would have a higher order in the set if and only if it has been composed in the hierarchy when the other UCA has existed. In other words, the the UCAs with less order in the set had probably been composed, or designed before the UCAs with higher orders. We define the function *order(U)* on the set of UCA defining the compositional order of each UCA. Also we define $\leq$ as the partial order of UCAs on the set of *order* of the UCAs. Therefore, if $order(U1) \leq order(U2)$, then $U1 \leq U2$.

9.5.2 Forward Propagation Algorithm

This order would help us preserve the compositional order of the insertion of UCAs into the hierarchy. From the order of insertion, we would be able to minimize the process of propagation of changes in the hierarchy. Based on this order we determine which UCA in the hierarchy and in the set of affected UCAs from the change would be affected with the change. Starting from the UCA node in the hierarchy which the change has happened, the propagation algorithm would propagate the change on the UCAs first with a lower order in the set of forward dependent UCAs which are affected by the change. It performs such task in a recursively on the forwardly defendant UCAs in the hierarchy, starting from the direct dependent of the changed UCA. Therefore, we suppose that there happened a change in the UCA U in the dependency hierarchy. And we define $direct(U)$ as the set of direct forward dependent UCAs of the UCA U. the propagation is done through the following algorithm:

```
ForwardPropagate(U):
    if isPropagated(U):
        return True
    else:
        Propagate(U)
        for all(U1,U2) in direct(U):
            if order(U1) ≤ order(U2):
                ForwardPropagate(U1)
                ForwardPropagate(U2)
        isPropagated(U) = True
```

The method *isPropagated(U)* returns true if the forward propagation of changes has been already been reflected on the UCA U, otherwise it returns false. Moreover, method *Propagate(U)* performs the propagation by the recomposition of the UCA U, using his already propagated parent base and referred UCAs. Using the *isPropagated(U)* method and the partial *order* function on the set of UCAs and following this recursive algorithm, we guarantee that when the *Propagate(U)* method is being called, both of its base and referred UCAs has already been propagated with the changes of their parents recursively. This algorithm would guarantee that no UCA would be propagated more than once in the hierarchy and hence, the efficient propagation of the changes on each forward dependent UCA in the hierarchy.

9.6 Forward Retrieve of Extension Points

Forward propagation of structural changes of UCAs toward the specification is achieved with the track of the composition expression forming a dependent UCA. As described, the change would be propagated according to the ordered hierarchy of the dependencies of the UCAs. When each UCA is updated with the propagated change, it comes up with a modified set of states and transitions according to the type and nature of the change. These states and transitions may have been used as extension points in the forward dependent UCAs to the current UCA in the hierarchical dependency chart. Therefore we need to keep track of these these extension points to assure the correct propagation of the change in the recursively dependent UCAs. Hence forward

updates on the extension points in the hierarchy of these expressions forming the UCAs should be performed to retrieving such tracks.

Some of these changes may cause ambiguities in the hierarchy. For example, if a transition which is used in its forwardly dependent UCAs as an extension point is removed in the UCA, the forward dependent UCAs would face ambiguities in the process of forward propagation. While we are in the process of propagating the change in the hierarchy, the composition would no more result in the same behavior and hence, might not be valid for the designer. In such cases the tool pops up the designer and asks him to resolve this ambiguity by choosing another extension point or simply remove the dependency and break the hierarchy from that point. Such re-localization in the extension points are necessary to maintain the process of forward propagation of the changes.

In other cases which the performed change in the UCA is not related to extension points used forward in the hierarchy, the update on the extension point is just limited to the updates on state names. The problematic issue for such process would be that these states may be renamed through the use of the UCAs in future compositions, and are impossible to retrieve. Therefore, we keep a characterization of each state in the composition expressions. This means that we keep a hash name for each state and its related transitions, which is generated automatically within the tool and could not be accessed or changed by the designer. This hash name acts as a signature for

that state, and is updated only after the UCA is propagated and therefore according to the recursive algorithm, its forward dependent UCAs are also propagated with the change. This trick differentiate between the presented labels for the states and labels in the automata. The hash name would be further used to recursively update the extension points during the forward propagation process. It is notable that this signature and its update is not visible to the designer and would never been observed by the user during the forward propagation process.

9.7 Conclusion

In this chapter, we presented the details of the idea and the implementation of an interesting feature in the tool on the traceability of the UCAs in the specification. We discussed how the composition of the UCAs and their dependencies would recursively form a hierarchical dependency chart, and how structural changes in each UCA would could be affective to its forward dependent UCAs. We detailed on how we implement this idea on a recursive algorithm, and shown how we use the partial order of insertions of UCAs in the hierarchy to make this process more efficient and prevent repetitive propagations. In the next chapter, we practice our composition and verification approach on an actual specification generation problem.

Chapter 10

Case Study: e-Purchasing System

10.1 Introduction

In this chapter, we show the applicability of our approach in UCOMV by applying it on a real-life specification problem. We present the application of our methodology on the elaboration of the behavior specification of an e-purchasing system. We show how the overall system specification is gradually formed from the initial requirements and their respective UCAs following several increments in the primitive specification. We also illustrate how the generated compound model of the requirement can be verified for the desired behavioral properties.

The chapter is organized as follows. Section 10.2 describe the early informal requirements of the system and show how the primitive specification is defined by the first set of UCAs. Section 10.3 defines the three steps of composition and the

composition expressions in each step which are to be followed to reach the target overall behavior model. Furthermore, Section 10.4 illustrates two cases of behavioral verification on the generated UCA model and discusses the result from the verification. Section 10.5 analyzes the results and concludes the chapter.

10.2 Partial Requirement Specification

The behavioral requirements of the e-purchasing system include a wide range of use cases on purchasing related activities and features. It covers all the activities from the early surfing to the final check out and payment in the process of purchasing an artifact in an online shopping store. We restrict our case study to the description of the system behaviors from the clients point of view. We present the informal description of partial requirements of such system from the clients perspective. From there, we build a primitive specification of the UCAs representing each separate functionality in the system and show how these informal requirements are transformed to our formal model of UCA as partial behavior models. These UCAs are then composed to form the requirements specification of the system which is subjected for verification over safety and liveness properties.

The following informal description of requirements express how a customer would enter to the system, orders his artifacts and pay, as well as some features and possibilities that he could benefit from. It basically expresses some scenarios which the

user could pass through the system in order to achieve his final goal of purchasing. This informal expression of requirements could be expressed from a stakeholder as the start point of a prospective system, and reflects what he has in mind from such a system.

```
The customer has to select a product. He either consults the
catalog list or makes a search with the name of the product in
the available catalogs, and then selects the product. Giving
the needed quantity, the availability of the product has to be
checked. Then, the order is placed and some information about
the type of delivery are asked to be specified. Finally, the
payment should be made. The customer should pass an
authorization process before the payment, and a quote of
report may optionally be printed after the order is placed.
```

As the first step, the informal requirements expressed for the system are partitioned into basic functionalities. As other approaches for the requirement analysis, this partitioning would be an state of the art from the analyst. It is notable that breaking the overall informal requirement specification is an important step in the requirement analysis, which can have a huge effect on the process of generating the overall specification and the composition strategy. A good partitioning of original requirements could help the process with clear composition and property analysis while a bad and overlapping partitioning could result in further difficulties in the study of final model. Hence the analyst can refer to his experience and other similar systems to the suite which may help on determining basic functionalities from the original overall requirement description.

In our example of e-purchasing system, the analyst may think of different partial behaviors, each in charge of a part of the requirements. For example the selection of the product and searching it could form an independent functionality. Also the general famous behaviors which are present in such systems could also be distinguished from the description and formally modeled as UCA. Formal UCA models in UCOMV are given in Figure 29 which will act as the initial partial behaviors and the primitive specification of our example. Different state colorings which has been selected for different UCA models can be seen in this figure. These colors can be further used to visually track each behavior in the final compound model.

10.3 Composition Increments

The initial UCAs are the initial step toward gradually forming the target specification. The overall model will be constructed in three composition steps. In the first increment, two new UCAs as Prod_Select_1 and Order_1 are created. The Prod_Select_1 UCA aims to insert the process of authorizing the client after the selection of the product and the Order_1 would place calculation of the price after the selection of the order. We note that these new behavior models are composed independently for different parts of the requirements. They will further be merged together and form the overall behavior model. The composition expressions for the two new UCAs are given as follows:

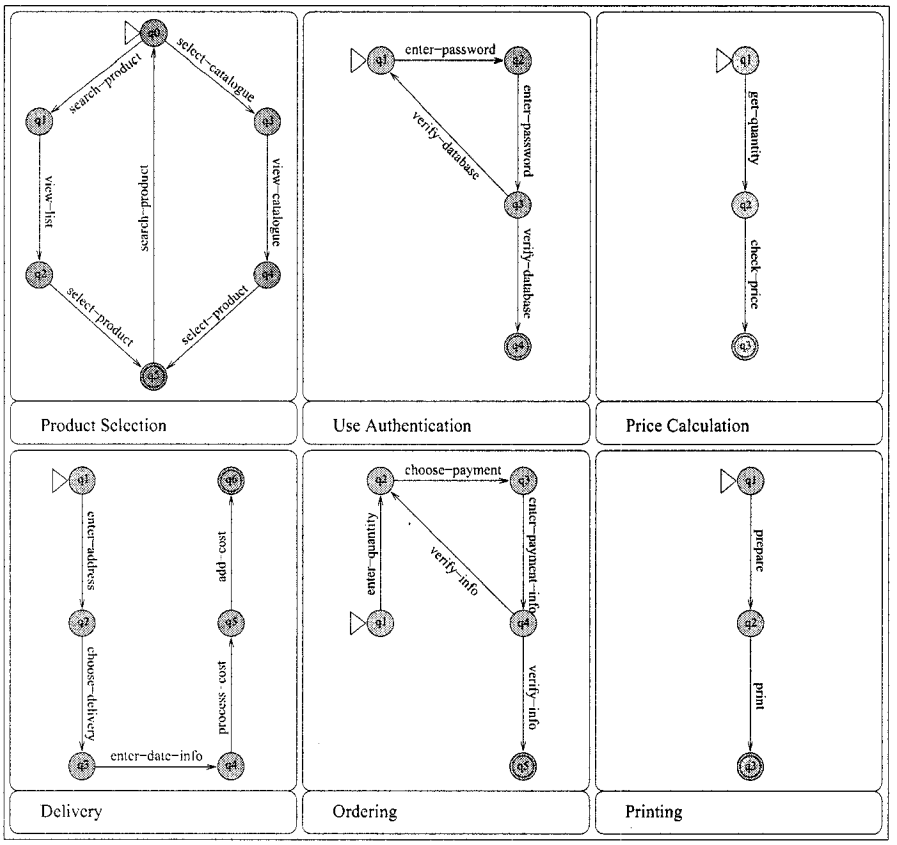

Figure 29: Primitive UCA specification of e-Purchasing system

1. Prod_Select_1 := ALTERNATIVE (Prod_Selection,Login) IN {q5}

2. Order_1 := INCLUDE (Order,Price_Calculate) AFTER {(q0,enter_quantity,q1)}

With the evaluation of the two specified composition expressions, the two new UCAs are added to the set of behaviors. In the second step, using the newly composed Order_1, a UCA will be generated which asks for the customer's delivery information before making the payment. We use UCA Order_1 generated in the previous increment. The transition (q6,check_price,q7) is the extension point of Order_1 where we want to include the Delivery UCA.

1. **Order_2 := INCLUDE (Order_1, Delivery) AFTER {(q6,check_price,q7)}**

Finally as the third steps, we generate the Order_3 UCA by adding a quote printing option for Order_2 UCA. Then we compose the generated Order_3 with Prod_Select_1 in order to gain the overall behavior in Prod_Select_2. The second composition expression in this step includes the whole generated behavior of the UCA Order_3 after the login in Prod_Select_1.

1. **Order_3:= EXTEND (Order_2, Printing) IN {q3}**

2. **Prod_Select_2 := INCLUDE (Prod_Select_1,Order_3) IN {q4}**

Covering all of the needed partial behaviors, Prod_Select_2 models the specified informal requirements of the e-purchasing system. The output of the UCOMV for this UCA after three increments is shown in Figure 30. The initial behaviors can be tracked with in Prod_Select_2 according to their initial color. The coloring of each state shows the behavior which formed that part of the compound model and hence, the overall structure of the initial behaviors can be tracked on the final product. The intermediate UCAs Order_1, Prod_Select_1, Order_2, and Order_3 are not shown here and are presented in Appendix A of the thesis. Prod_Select_2 is the product behavior of the process and is subjected to model verification over desired behavioral properties. It is transferred to the verification environment for behavioral property verification.

10.4 Behavioral Verification

The analyst can perform a wide range of behavioral verifications in terms of liveness and safety properties, and further analyze the correctness of the composed model for the decision on possible revisions in the requirements and their relations. For the e-purchasing requirement model of Prod_Select_2, we define an example of a behavioral verification which is modeled as a liveness property in the standard LTL template of precedence in the tool. According to the specified requirements, after the selection of a product, the authorization of the user should always be checked before the payment is done and the purchase is finalized. The analyst want to make sure that if the product is selected and the delivery information is entered, eventually the payment is done before the system terminates on the delivery. Therefore the following LTL property is defined for the tool to be verified on the generated model:

$$[\,]\ (\textbf{select-product} -> (\textbf{user-authorized U payment-verified}))$$

According to the three composition steps we performed, considering the semantics of the operators used, the analyst can be sure that this intended property will be verified on the generated model. However performing of the verification on the Prod_Select_2 UCA model in the first step would bring us to a certain scenario which prevents the model to verify the property. This scenario happens because of the loop in the Prod_Select UCA, and would be detected by the model verification module, interpreted and visualized on the Prod_Select_2 UCA. The trace resulting in the

deadlock can be followed in the model as search-product $->$ view-list $->$ select-product. This loop is an infinite behavior which prevent us to verify the property, and hence is a behavioral breach over the intended property.

As checking the safety compliance of the generated model, we give an example of checking the authorization of the user. The analyst would want to make sure that always in the system, the payment would not be completed before the user enters his address. The negation of this behavioral property would result in that the payment is not done until the user enters his address. Therefore we would have the following formula on the property manager component:

$$[\,]\ (\neg\ \textbf{choose-payment U enter-address})$$

As we can see in the generated model, this property is verified on the models. This compliance can be expected according to the compositional semantics of the Include operator which is used to compose the Delivery behavior inside the Ordering. This semantic would guarantee the insertion of the traces of the Delivery behavior inside the Ordering, and therefore the enter-address behavior would be placed before the choose-payment in in all the scenarios in the composed model in the second composition step. As we can see, the compositional semantics of the operators could guarantee the intended behavioral semantics from the original requirements if they are used properly in accordance to the target behavior.

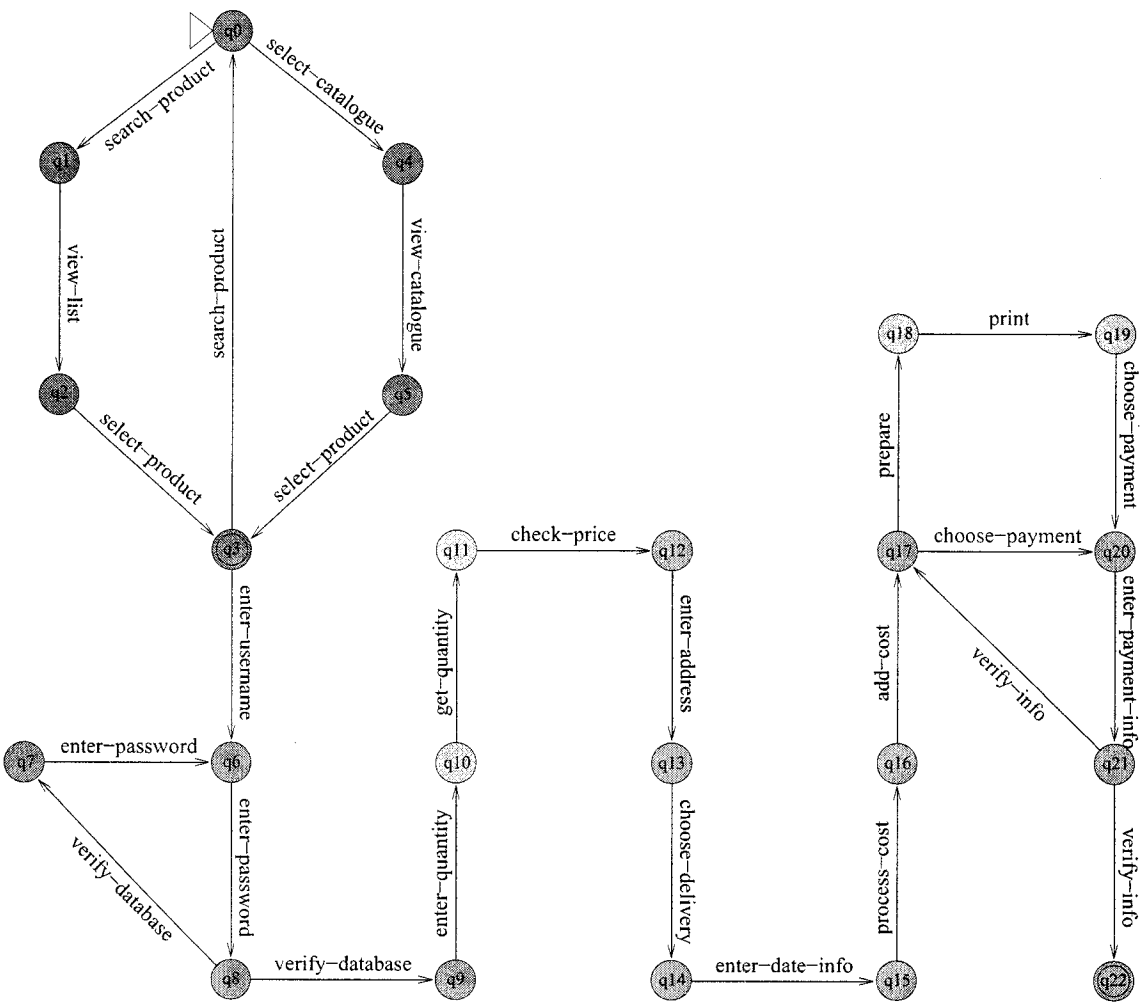

Figure 30: Product Selection 2 after three increments

10.5 Conclusion

In this chapter, we saw the practice of our approach in UCOMV to help an actual specification generation problem. We show that as the UCAs grow into complex models through composition increments in the specification, the space of the traces also grows and defining appropriate properties for the checking of their semantics gets more complex. Moreover the classical problem of exponential state growth to the order of the size of the UCA will increase the complexity of the verification. We illustrate how the designer can reduce this complexity into smaller-scale verification problems on each increment through the generation of the specification for the requirements of an actual system.

Chapter 11

Discussion, Conclusion and Future Works

The motivation for the work presented in this thesis is to provide a requirement analysis approach and a tool support for formal modeling, composition and verification of the requirements as Use Case Automata. The framework supports expression of new models, their automated composition, and the verification of the generated models for checking of model compliance to the desired behavioral properties. An incremental approach for the generation of the requirement specification is supported in UCOMV tool. The easy-to-use specification design interfaces with useful features along with the composition engine and the verification module in UCOMV, backed by a rigorous theoretical work, provides a tool for elaboration of state-based requirement specifications.

The formal model of requirements presented in this thesis is based on scenarios and Use Case Automata as partial narrations of system requirements, which is widely undertaken and accepted in the literature and industry. We provide a formal model and an incremental composition and verification methodology for gradual elaboration of requirement specifications. We also provide a tool support with an easy-to-use graphical interface for requirement presentation, composition and verification. The theory for the modeling and composition is based on earlier works in [32, 33, 24] and [31]. Many interesting features to facilitate the design and composition of requirement models have been provided in the tool.

Our approach of generating requirements specifications consists of composition UCA expressions in an incremental manner. The approach starts with defining a set of initial UCA models for basic behaviors in the system. Furthermore in each increment, the specification is enriched by the evaluation of composition expressions, and new behaviors are generated and added to the existing set of models. Generating a new UCA does not constrain the two involving base and referred UCAs, which is a significant difference of our approach from existing approaches on use case composition. The generated model is guaranteed to carry the behaviors aimed from the composition operator and its semantics, and nothing more. The new UCA is subjected to behavioral verification for the desired properties in the verification layer of the tool provided as an integrated framework for checking of model compliance to temporal safety and liveness properties.

The dependency relation of the new UCA is retained in a recursive hierarchy of composition. This hierarchy is used to trace and propagate the structural changes in UCAs to all of their forward depending UCAs in the hierarchy in an optimal manner. This provides a traceability features for automated reflection of structural changes in the compound requirements, an essentially useful feature for the maintenance of the requirement specifications. Moreover, the generic design of the compositional semantics can be used to introduce new composition operators. The extendability of the UCOMV tool with new composition operators within a XML based schema opens a wide ground for future extension of the tool to user-defined operators with intended customized semantics for synthesizing the behavioral models at state machines level.

Incremental generation is a key issue in our approach. As the UCAs grows into complex models through composition increments in the specification, the verification of the generated model becomes an intensively complex task. In our incremental approach, the analyst has the ability to break this complexity into smaller-scale verification problems on each increment. Starting with small initial models and breaking the generation of large UCA models into more increments would help the analyst to handle this complexity and perform proper verifications on the behaviors of the requirement models, and hence, generating more trustable and consistent specifications.

Although theoretically our approach has no limitations in terms of the complexity of UCA models, some limitations may apply to the size of the generated UCA models for the composition and verification. This stems from the limitations on the higher complexity orders of growth of the number of states generated during the composition and verification process. As the size of the UCA models grow linearly in terms of the number of states, the complexity of the composition grows in a polynomial order while the complexity in the verification side grows in exponential terms. This brings the classical problem of state explosion on the verification side. The physical limitations of the computers may also prevent us from the study of arbitrarily large and complex systems. However, despite these theoretical and implementation considerations, the system performs very well on regularly large system descriptions in real life. We have tried to generate large enough systems to detect this realistic limitations, and came up with a high performance on system descriptions with up to 400 states. This is merely very high consideration and could easily cover the large scale industrial specification generation problems.

As major extensions to the work, several ideas has been seen both in the theory and tool support of the approach. As in the theoretical side, we seek to develop the idea of deducing a given formal model from the existing models and compositional semantics. This means an up-side down approach comparing to the method presented here, and aims at partitioning the an overall given system model to smaller deduced models using the semantics of the operators, and needs a rigorous theoretical study on

the automated deduction of the behaviors. We also intended to study the persistence of the verified properties on UCAs in their composition products, which we call as property inheritance. As in the tool side, we see to present UCOMV as a plug-in for Eclipse. Providing an interface for the migration of scenarios from other scenario notations (such as MSC) to UCA is seen as a future extension to the tool. Also the rewriting of the parsing and verification rules in the verification layer to support other model checkers to the suit such as UPPAAL is seen as a part of the future work. The tool is available for download at: *http://users.encs.concordia.ca/~s_kolahi/UCA/.*

Appendix A

Intermediate UCAs

In this section, the intermediate UCAs generated during the increments in the UCA specification of the e-purchasing system in Chapter 10 are given. The two composition expressions in the first increment generate the two UCAs of Prodselect1 and Ordering1 while in the second increment, Ordering2 is composed and finally in the third increment, the final UCA model Ordering3 is generated as the overall model representing the given informal requirements of the e-purchasing system.

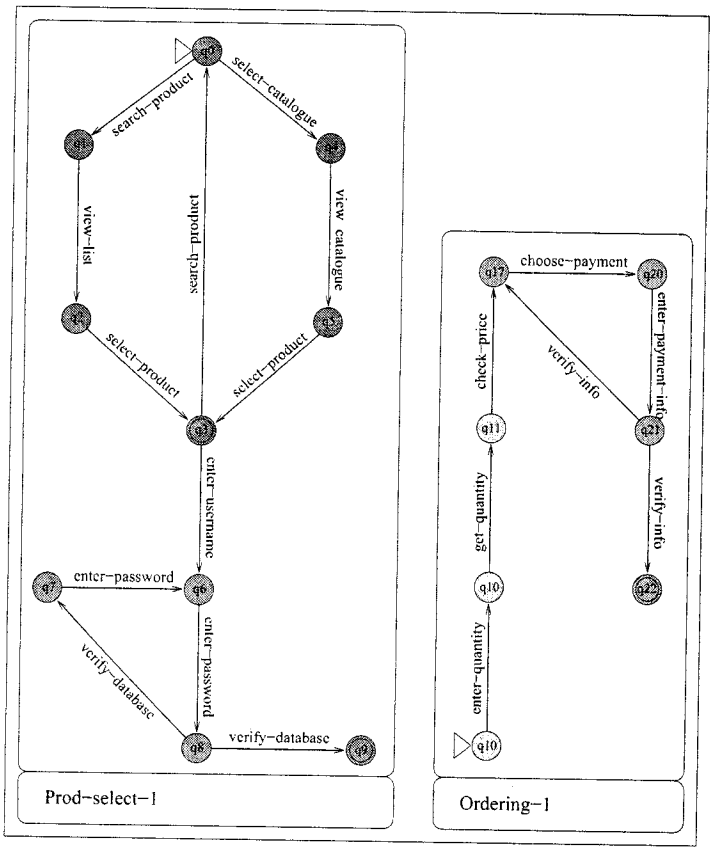

Figure 31: Prod-select-1 and Ordering-1 UCAs

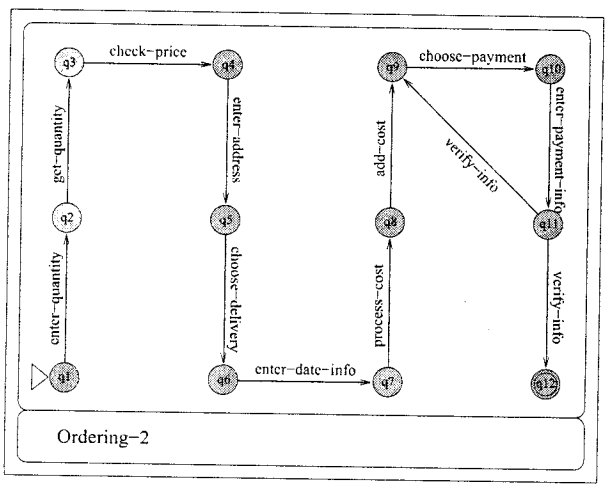

Figure 32: Ordering-2 UCA

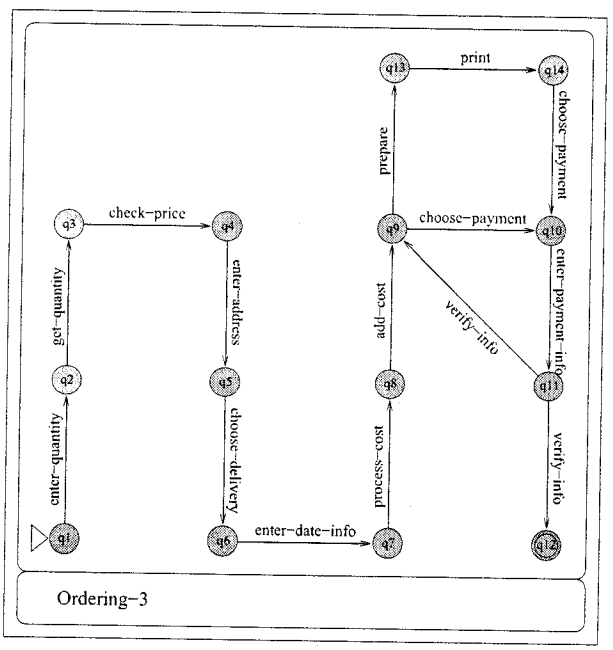

Figure 33: Ordering-3 UCA

Bibliography

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, 2003.

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

[4] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 35–48, 1996.

[5] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR)*, pages 114–129. Springer-Verlag, 1999.

[6] D. Amyot, W. D. Cho, X. He, and Y. He. Generating scenarios from use case map specifications. *Third International Conference on Quality Software (QSIC'03)*, November 2003.

[7] D. Amyot and A. Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommunication Systems*, 24(1):61–94, 2003.

[8] Daniel Amyot. *Specification and validation of telecommunications systems with use case maps and lotos.* PhD thesis, University of Ottawa, 2001. Adviser-Luigi Logrippo.

[9] J. Araujo, J. Whittle, and D. K. Kim. Modeling and composing scenario-based requirements with aspects. In *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE)*, pages 58–67. IEEE Computer Society, 2004.

[10] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice inmessage sequence charts. In *Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 259–274. Springer-Verlag, 1997.

[11] F. Bordeleau and J. Corriveau. From scenarios to hierarchical state machines: A pattern based approach, 2000.

[12] F. Bordeleau and J. P. Corriveau. On the importance of inter-scenario relationships in hierarchical state machine design. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 156–170. Springer-Verlag, 2001.

[13] F. Bordeleau, J. P. Corriveau, and B. Selic. A scenario-based approach to hierarchical state machine design. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, page 78. IEEE Computer Society, 2000.

[14] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45 – 80, 2001.

[15] D. A. Peled E. M. Clarke, O. Grumberg. *Model Checking*. The MIT Press, 2000.

[16] H. Foster, J. Kramer S. Uchitel, and J. Magee. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 152–162, 2003.

[17] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Tool support for model-based engineering of web service compositions. In *IEEE International Conference on Web Services (ICWS)*, pages 95–102. IEEE Computer Society, 2005.

[18] M. Glinz. An integrated formal model of scenarios based on statecharts. In *Proceedings of the Fifth European Software Engineering Conference*, 1995.

[19] M. Glinz, S. Berner, S. Joos, J. Ryser, N. Schett, and Y. Xia. The adora approach to object-oriented modeling of software. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, pages 76 – 92. Lecture Notes In Computer Science, 2001.

[20] P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems: An Approach to The State-Explosion Problem*, volume 1032. Springer-Verlag Inc., 1996.

[21] D. Harel and H. Kugler. Synthesizing state-based object systems from lsc specifications. In *Revised Papers from the 5th International Conference on Implementation and Application of Automata (CIAA'00)*, pages 1–33. Springer-Verlag, 2001.

[22] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[23] ITU. Message sequence chart (msc). Recommendation Z.120, 1999.

[24] Siamak Kolahi, Aziz Salah, Rabeb Mizouni, and Rachida Dssouli. Tool support for composition and verification of formal behaviors. In *4th IEEE International Conference on Innovations in Information Technology (Innovations07)*, 2007.

[25] Peter Linz. *An Introduction to Formal languages and Automata*. Jones and Bartlett Publishers, third edition, 2000.

[26] M. Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1):529–554, 2003.

[27] M. Lohrey and A. Muscholl. Bounded msc communication. *Information and Computation*, 189(2):160–181, 2004.

[28] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs.* John Wiley and Sons Ltd, 1999.

[29] E. Mäkinen and T. Systä. Mas: an interactive synthesizer to support behavioral modelling in uml. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 15–24. IEEE Computer Society, 2001.

[30] R. Marelly, D. Harel, and H. Kugler. Specifying and executing requirements: the play-in/play-out approach. In *17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '02)*, pages 84–85. ACM, 2002.

[31] Rabeb Mizouni. *Formal Composition of Partial System Behaviors.* PhD thesis, Department of Electrical Engineering, Concordia University, 2007. Adviser - Rachida Dssouli and Aziz Salah.

[32] Rabeb Mizouni, Aziz Salah, Siamak Kolahi, and Rachida Dssouli. Composition of use cases using synchronization and model checking. In *26th IFIP International Conference on Formal Methods for Networked and Distributed Systems (FORTE)*, pages 292–306, September 2006.

[33] Rabeb Mizouni, Aziz Salah, Siamak Kolahi, and Rachida Dssouli. Merging partial system behaviors: Composition of use case automata. *IET Software*, 2007.

[34] OMG. Unified modeling language (uml). Specification Version 2.0, 2000.

[35] J. Ryser and M. Glinz. Dependency charts as a means to model inter-scenario dependencies. In *Modellierung 2001, Workshop der Gesellschaft fr Informatik*, 2001.

[36] T. W. Finley S. H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Sudbury, MA, 2006.

[37] S. Some. Specifying use case sequencing constraints using description elements. In *Proceedings of the Sixth International Workshop on Scenarios and State Machines (SCESM)*, page 4. IEEE Computer Society, 2007.

[38] S. Some and R. Dssouli. An enhancement of timed automata generation from timed scenarios using grouped states.

[39] S. Some, R. Dssouli, and J. Vaucher. From scenarios to timed automata: Building specifications from users requirements. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 48. IEEE Computer Society, 1995.

[40] Bertrand Tavernier. Calife: A generic graphical user interface for automata tools. *Electronic Notes in Computer Science*, 110:169–172, 2004.

[41] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. Ltsa-msc: Tool support for behavior model elaboration using implied scenarios. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 597–601, 2003.

[42] S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *23th International Conference on Software Engineering (ICSE)*, 2001.

[43] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *European Software Engineering Conferece and ACM SIGSOFT Foundations of Software Engineering (ESEC/FSE'01)*, 2001.

[44] S. Uchitel, J. Kramer, and J. Magee. Negative scenarios for implied scenario elicitation. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT/FSE)*, pages 109–118. ACM Press, 2002.

[45] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29:99–115, 2003.

[46] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13:37–85, 2004.

[47] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, pages 429–528. Springer-Verlag, 1998.

[48] N. Kicillof W. Grieskamp. A schema language for coordinating construction and composition of partial behavior descriptions. *5th International Workshop on Scenarios and State Machines (SCESM'06)*, May 2006.

[49] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *International Conference on Software Engineering (ICSE)*, pages 314–323, 2000.

[50] J. Whittle and J. Schumann. Statechart synthesis from scenarios: an air traffic control case study. In *Workshop on Scenarios and State Machines at ICSE*, 2002.