FORMAL COMPOSITION OF PARTIAL SYSTEM BEHAVIORS

Rabeb Mizouni

A thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Concordia University

Montréal, Québec, Canada

December 2007

© Rabeb Mizouni, 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-37755-0
Our file Notre référence
ISBN: 978-0-494-37755-0

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Formal Composition of Partial System Behaviors

Rabeb Mizouni, Ph.D.

Concordia University, 2008

Modeling the behavior of a system under development has shown to be a very effective way to ensure that it will have better chance to be constructed correctly. With the growing complexity of systems, building this model has become a major task that requires a significant time investment and a high level of expertise. Incremental approaches that help construct a system model from partial behavioral descriptions have been widely adopted. The challenge in such approaches lies in finding both the adequate behavioral formalism that fits the needs of the analyst as well as the formal composition mechanism that facilitates the generation of the expected behavioral model and produces a verifiable model.

Within this framework, use case approaches have been accepted in the industry because they make the process of requirements engineering simpler. In the first stage of their development, use cases have been associated with requirements gathering and domain analysis since they allow the partial description of system behavior in a more intuitive manner. During the last decade, their use has been expanded to include all phases of the lifecycle. Consequently, the model representing use cases has an increasing importance.

Although use case approaches present benefits in terms of improving the communication among stakeholders, permitting incremental construction of the system specification, and

improving the requirements traceability, such approaches have some drawbacks in relation to their lack of formality. In fact, it is difficult to validate and verify use cases for completeness and consistency.

This thesis addresses the problem of modeling use cases and their composition based on formal models in order to generate a system specification that can be used for validation and verification. We tackled the problem of both composing overlapping use cases that share partial similar behaviors, and composing non overlapping use cases according to additional criteria.

We experimented with different formal models of use cases having different levels of expressivenesses to develop an approach for use case composition. All use case models we tackled are state-based models. We first started by proposing an approach for composing use cases expressed as extended finite state machines with variables that characterize their states. The use case model allows the definition of explicit loops. The state characterization is used as the criterion of composition. It allows the detection of common states between use cases that have to be merged in the overall system model. When composing, we proposed an approach that protects the user-defined loops from unexpected scenarios that may threaten their behavior.

As a second step, we proposed to compose use cases based on the interactions they are making between each other. In this context, an interaction is defined as an invocation of a use case by another. Unlike the first approach, use cases are no more considered overlapping. When composing, we developed an approach that avoids unexpected scenarios.

Finally, we proposed a general approach for composing system behaviors where partial system behaviors are defined as state based model using imperative expressions. Each

use case describes a certain system concern. The imperative expression represents the composition criterion. In fact, it defines the semantics of the composition to perform. Our approach is fully automated and provides the advantage of generating a state based model that meets the intended behavior without allowing unexpected scenarios. The approach is formalized in the case of finite state machines and extended finite state machines.

Acknowledgments

Many people were involved in the production of this thesis, and I wish to express my gratitude toward them.

I would like to show my gratitude to *Rachida Dssouli*, my supervisor. I have been most fortunate to enjoy her wisdom, experience, guidance, and also her warm encouragement when needed the most. I also want to thank my co-supervisor, Aziz Salah, for his support, discussions, feedbacks, permanent help, and availability he offered me during this thesis. I extend my thanks to my committee, Dr. Patrice Chalin, Dr. Mourad Debbabi, Dr. Ferhat Khendek, and my external examiner, Dr. Nadia Tawbi, for gracefully accepting to review and comment this work.

I thank my colleague, Siamak Kolahi, who did the implementation of the tool. His collaboration in validating the approach was of a great help. I thank my friends, Abdelghani Benharref and May Elbarrachi, for the several discussions we had, for sharing their technical knowledge and their humor with me. I am indebted towards Adel, Dalia, Lamia, Salam, Syrine, Yousser, Zaki... for supporting me in the hard moments and for being my family in Montreal.

My deepest thanks want to my beloved husband, Anis, for the sacrifices he did in order to allow me finishing this work. Without his help, confidence, permanent encouragements, and belief in me I would never finish this thesis. Finally, I want to thank my sons, Haroun and Adam, for the moments of happiness they offered me. I love you all!

Contents

List of Figures

List of Tables

Dedications

À Ma mère qui m 'a poussée à réaliser ce rêve...

À mon père qui m'a offert le support moral et affectif tout au long de mes études...

Chapter 1

Introduction

1.1 Motivation

Requirements engineering and design remain the critical phases in system development process. They are the phases where the needs of the customers have to be unambiguously specified in order to ease the process of getting the right system. Use-case based approaches are one of the techniques used in requirements engineering. Over the years, they have gained a lot of importance in the system development life cycle to become a widely used method in the industry for elaborating requirements of reactive systems.

Behavior modeling plays an important role in the engineering of software systems. It allows the development of systematic approaches to requirements capture, specification, design, simulation, code generation, testing, and verification. Several notations, techniques, and tools that support behavior modeling already exist for simple aspects.

However, as systems are becoming larger and more complex, systematic generation of a formal behavioral model from informal requirements has become a crucial and challenging task. A faulty model can lead to a low-quality system and can increase considerably the cost

of the system. Many studies have been conducted in order to formalize this model generation process [39, 51, 11, 12, 109, 105, 95]. They aim at generating a state based model of the overall system behavior. This model is intended for the validation and the verification of the user requirements. Moreover, incremental approaches seem to be appropriate for the generation of a formal behavioral model of the system. They help not only in the process of building the system behavior in an iterative way but assist also in the process of validating and verifying the user requirements. To accomplish this goal, use case and scenario based approaches have been well received and have a wide acceptance in the industry.

Intuitively, a use case is a sequence of actions performed by the system to yield an observable result of value to a particular user, and a scenario is an execution of a use case. In such approaches, the system functionalities are described by a set of use cases that are more or less independent of each other. Primarily, use cases have been associated with requirements elicitation and domain analysis. However, the scope of use cases has broadened to include modeling constructs at all steps. Due to this expanded scope, the representation of use cases has an increasing importance.

Use case approaches have many advantages. In fact, partially describing the reactive system behavior is less difficult than specifying it as a whole. It makes the requirements elicitation more intuitive and the communication between stakeholders more straightforward. In addition, use case approaches promote the incremental construction of the behavioral model and help in its maintenance because they facilitate the traceability of the requirements in any phase of the system's development lifecycle.

Despite these advantages, use case approaches present some drawbacks that are closely related to the lack of formality. Use cases are descriptions that are expressed in most cases

using informal languages, which are usually ambiguous and error prone. In addition, such informal descriptions are difficult, not to say impossible, to use in a systematic approach for synthesizing an overall system specification that can be validated and verified against completeness and consistency. To accomplish this task, the dependencies among use cases must be analyzed before their composition. This last activity is complex and may be facilitated by the usage of formal methods based on a formal model of use cases such as state based models.

1.2 Problem Statement

Nowadays, the specification of individual use cases is achieved in natural languages such as English, which provides ample room for miscommunication and misunderstandings. Use cases provide a much less formal specification of their instances (i.e., individual usage scenarios), a fact that makes the relationships defined among them (provided by extends and uses associations in the case of UML) not well defined.

While everything may seem clear in the highest level of abstraction, the translation of use cases into design and code at lower levels of abstraction is based on informal human understanding of what must be done, which makes such activity error prone. It also causes problems when it comes to utilize use cases for the specification of acceptance tests and for the validation and verification, because the criteria for passing those tasks may not be adequately and unambiguously defined. Consequently, formal models with well defined semantics to represent use cases have been suggested. Approaches [12, 62, 102, 89, 26] and tools [93, 18, 4, 82, 100] that support the description of use cases and their composition for a construction of a system specification have been developed. Such approaches have different

models of use cases with different level of expressiveness. The generation of a system specification is achieved in different level of automation. It can insure the correctness of the derived model by construction when the process is fully automated or by additional verification when the process is semi-automatic or manual.

To overcome the informality of use cases, many studies have suggested describing use cases as state based models [39, 109, 92]. Such modeling concentrates on the internal states of individual components. State-based formalisms are widely used in requirements engineering, particularly in distributed and real-time system design. In many cases, these formalisms are the basis for rigorous automated analysis (such as model checking), laborious analysis, and formal verification of the proposed system's behavior. Despite the benefit of having such models early in the development lifecycle, the direct use of state-based models is not widespread in the industry. Use cases must be modeled as FSM/EFSM then composed to derive the model representing the overall system, a demanding and challenging task that depends on several factors:

- **The use Case Model Expressiveness:**

  The choice of the use case presentation is a decisive step in any approach of use case composition. It illustrates the level of details about the system functionalities available at the time of modeling, but it is also where the complexity of the composition is implicitly decided. Having an expressive model for the description of partial system's behavior helps the analyst to describe a more realistic specification, that is closer to the design phase. Depending on the application and the phase of development lifecycle where the validation and verification are performed, the expressiveness of the use case model plays an important role in the validation and the verification tasks.

However, it makes the composition harder. In the opposite, having a simple use case model makes the composition simple but generates a high-level specification, a fact that decreases the guarantees of correctness [106]. Consequently, there is a trade off to make in order to balance between the model expressiveness and the composition complexity, depending on the intended use of the generated model.

- **The Automation Level of the Composition Approach:**

  Automation is another challenge that faces the generation of a system specification from use cases. Very often, during the requirements engineering phase, the requirements are not well defined and subject to modifications. Having an automated approach at this stage makes the task of the modeler much easier. In addition, a realistic and reliable specification of the overall behavior cannot be obtained unless performing several increments. Many researchers [86, 13, 12, 57, 101] have opted for an automatic generation, others choose the semi-automatic or the interactive [32, 88, 23, 23, 38, 98, 97] generation of system's specification. Each has advantages and limitations. But the choice will mainly depend on the objectives of such composition. If we take the case of generating code for rapid prototyping, then the automation will be a more appropriate way since the prototype itself will be improved progressively [46]. Nevertheless, such approach seems inadequate if the purpose is to detect or to avoid the implied scenarios, inconsistencies and completeness of the specification. In such case, interactivity with the user will give the potential of accepting (or not) an implied scenario and hence completing the specification by possible but forgotten scenarios.

- **The Formality Level of the Model Representing the Partial Behavior:**

  Formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems [27]. Formal methods may be used to give a description of the system to be developed in the desired abstraction level. This formal description can be used to guide further development activities. In use case based approaches, having a formal model of the requirements allows their validation and verification. It also allows to define automated approaches for use case composition as specified previously.

  The application of formal methods is considered as the basis of rigorous systems development. However, the effective exploitation of these methods encounters several problems that may be explained by the increasing complexity of the applications, the lack of users with appropriate skills and background, etc. Usually, the formal representation of use cases is not not always intuitive and easy to understand by the different stakeholders. It requires an expertise from the stakeholders which explain their limited use in industry.

  The difficulty in formalizing informal use cases lies in defining a formal but still intuitive use case model to benefit from the advantages the formality brings to the composition approach, without making the specification of the initial set of use cases a hard task for the modeler.

- **Conformity to Initial Behavior:**

  When composing use cases, it is frequent that some scenarios that are not expected appear in the resulting global model. They are in fact scenarios that were not specified in the original use cases and appeared because of the use case composition. We

call them *implied scenarios.* These scenarios may result from inconsistencies in the specification, but also may show possibly desired but forgotten scenarios. In the latter case, implied scenarios are considered as a positive behaviors since the system is supposed to exhibit them. Undesired implied scenarios are considered as negative behaviors and have to be removed [102].

- **Classification of Composing Partial Behaviors:**

  Since use cases are in fact designed and written independently, it is not always easy to merge them. In the following, we present some composition approaches.

  1. Implicit Composition: in this approach, scenarios are integrated without any additional constraints. It allows the definition of overlapping use cases. This overlapping is in fact used as point of merger where the use cases will be composed. Often, implicit composition uses the characterization of the states in different use cases in order to detect the points of merger. In addition, a scenario (which represents a use case run) represents a scenario of the system, starting from an initial state until reaching one of the final states of the system, as shown in Figure 1. Hence, the system behavior is described as a set of traces. The advantage of this method is the optimization of the possibilities in each system state. It means that for each represented state, the specification gives all the possible behaviors that the system may execute from that state. However, this approach increases the number of implied scenarios (resulting from the composition).

  2. Explicit composition (also called composition-based approaches [63]): not only use cases are given but also the order in which they should be composed is

specified. Hence, in such an approach, use cases can be sequential, alternative, parallel or iterative. The usage of composition notations is a way to express the desired order of use cases with the possibility of constraining the execution of use cases with pre-conditions. We also note that this mode of composition has the advantage of protecting individual use cases by reducing the risk of having implied scenarios to the connection points where the use cases are grafted.

It has been noticed that explicit integration of use cases is a more efficient way of modeling than implicit integration [63], because of the human effort necessary to identify the same states in the latter one. It also presents a different way of modeling the system behavior. While in the first case, overlapping between use cases is allowed, in this case it is not tolerated. Use cases has to represent different requirements with well defined functionalities. In fact, it encourages the separation of concerns into distinct features that overlap in functionality as little as possible. A possible system trace in this case would result from the composition in the specified order of some use case traces.

Using one of the composition scheme imposes some additional constraints. As an example, when using the explicit composition mode, Glinz [33] has imposed that all the scenarios should have one initial place and one final place. This is not the case when composing implicitly scenarios. However, while the first mode may reduce the number of implied scenarios, the second may deteriorate the desired behavior by introducing a large number of unspecified behaviors, and then algorithms for detecting them have to be developed.

- **Traceability:**

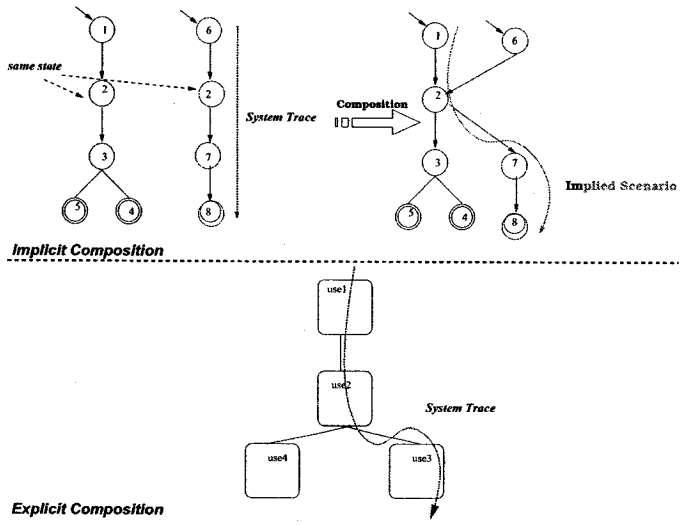


Figure 1: Implicit Vs. Explicit Composition

Since at early stages of the system requirements, the needs of the user may change, it is important to reflect these changes on the overall specification. Such modifications have to be made with the minimum cost. Traceability of requirements is defined as: *" the ability to describe and follow the life of a requirement, in both a forward and backward direction, i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases."* [35]

Use-case driven approaches can provide a natural vehicle that assures the traceability of functional requirements [87]. When functional requirements are both traceable and visible throughout essential development activities, the likelihood that required functionality will be accommodated in the delivered system as well as the quality of the software are improved. Hence, when composing use cases, it is of interest to have a parallel mechanism that keeps track of the original requirements from where a behavior is derived. Changes of the original use cases are then easier to reflect on the

system specification in one hand. Moreover, the back-tracking of the requirements to change, if the resulting system behavior is different from the modeler expectations, is more straightforward.

1.3 Thesis Contributions

Current use case composition approaches suffer from a lack of use case expressiveness and rigorous formality. Often, the model used for use cases does not handle data. Finding both an adequate model covering the needs of the analyst and a formal composition mechanism serving the generation of the expected overall behavior is challenging. Within this thesis, use cases are defined as state based models. Different formal state-based models are considered in order to study the expressiveness, the complexity, and the applicability of the composition in each model.

Our objective is to define a formal and automated approach to compose use cases taking into account these criteria:

- The model of use cases must capture the system requirements in a comprehensive manner. Hence, it needs to be intuitive and easy to use and to understand. Additionally, the model needs to give more expressiveness power to the analyst comparing to what already exists in the state of the art by handling data information.

- The model of use cases has to be formal in order to achieve the automation of the approach.

- The approach should help generating the overall state-based system specification at an early stage of the system development lifecycle.

- Implied scenarios may harm the behavior of the overall system. Often, when composing manually use cases, an interaction with the analyst is allowed in order to make decision about implied scenarios. In our case, we advocate an automated approach, fact that makes such interaction impossible. Hence, we target to reduce implied scenarios when composing use cases.

- The approach should support incremental elaboration of a formal system specification because the requirements are prone to change.

- The approach should help the maintenance of the system specification because changes are possible at this level of system description. Hence, the traceability of the requirements is of particular interest.

This thesis offers a set of contributions in terms of formal and automated partial behavioral description and composition, mainly related to implicit composition and explicit composition of use cases.

1.3.1 Implicit Composition of Overlapping Use Cases

Beyond simple models, the difficulty is to take into consideration variables in the composition, for which no composition approaches exist. We target the use of a state based model to represent use cases. However, a search for an appropriate model (new variant of models) that helps the issue of composition is essential. In a first stage, we tried to define a formal model of use cases based on a variable characterization of the states. The use cases are represented by a variation of EFSM (Extended Finite State Machines). We mainly enriched the model in [91] to handle loops and variables. We considered the explicit specification of loops in the description of use cases, which is different from the other approaches in the

state of the art [10, 109, 101, 91]. Use cases may contain loops that have to be protected while composing in order to avoid destroying their behavior by the introduction of implied scenarios.

1.3.2 Explicit Composition with Implied Scenario Removal

As a second step, we tackled the problem of explicit composition of use cases. By explicit we mean the use of interactions in order to compose use cases rather than using the characterization of the states. Use cases are defined as a variation of EFSM enriched with interactions. An interaction is defined as a call of a use case to another use case when performing an action. Consequently, we have allowed the specification of an interaction within the specification of the label of a transition. Each of these interactions defines a pattern that reflects the semantics of the interaction in the state-based model. After defining the different use cases, we generate the overall system specification using these state-based patterns applied directly to the set of use case in the form of a cut and paste operation. In order to avoid implied scenarios, we propose to generate a graph of interaction of the overall defined use cases. From this graph, we determine the possible interferences between use cases that lead to implied scenarios. Finally, additional constraints are added to the generated state-based model in order to remove such implied scenarios.

1.3.3 Explicit Composition of Use Cases using Imperative Expressions

Inspired by programming languages, we defined a new approach of explicit composition of use cases based on the definition of *imperative expressions*. These expressions allow the separation between the definition of use cases and the definition of interactions and specify the semantics of the composition the modeler asked for. We applied our approach in the

cases of finite state automaton and extended finite state automaton. This approach has the advantage of defining a formal mechanism of composition that replaces the cut and paste mechanism we proposed previously in section 1.3.2.

1. **Composition Approach with Imperative Expressions**

   The approach consists of specifying explicitly and incrementally composition expressions, rather than defining the interactions within the use cases, as we proposed previously. Each composition expression specifies the use cases to merge, the operator to use, as well as the place where to merge (called extension point). The result of the composition expression is a new behavior. It is derived from the merging of the two use cases, according to the semantics of the composition operator. Our approach is applicable for any state based model representing use cases. It allows the generation of a system specification in the same state based model of the original set of use cases. The different steps for such composition are described.

2. **Use Case Automata Composition**

   The formalization of the composition approach with imperative expressions in the case of Use Case Automata (UCA) is given. UCAs are finite state automata. They represent an intuitive and expressive way to describe partial system behaviors. An extension point can be either a state or a transition. Our approach handles the fact of inserting a use case in multiple extension points.

3. **Use Case Extended Automata Composition**

   The more detailed is the specification, the greater are the guarantees of correctness, [106] . Therefore, it is interesting to generalize our approach in the case of Extended Finite state machines. Hence, we propose to formalize the approach of composing use

cases using imperative expressions in the case of EFSM, we call Use Case Extended Automata (UCEA). In such specification, we distinguish between two kinds of variables: UCEA and specification variables. The first are considered local to the UCEA where they are defined, while the second they are global, shared by the set of use cases defining the specification.

1.3.4 List of Publications

1. Rabeb Mizouni, Aziz Salah, Siamak Kolahi, and Rachida Dssouli: *Composition of partial system behaviors*, IET Software Journal (formerly IEE Proceedings Software), Volume 1, Issue 4, August 2007.

2. Rabeb Mizouni, Aziz Salah, and Rachida Dssouli: *Using Formal Composition of Use Cases in Requirements Engineering*, The Nineteenth International Conference on Software Engineering and Knowledge Engineering SEKE'07, Boston, USA.

3. Rabeb Mizouni, Aziz Salah, Rachida Dssouli, and Siamak Kolahi: *Incremental Extended Use Case Composition* , 7th International Conference on New Technologies of Distributed Systems Morocco, Marrakesh, June 4-8, 2007.

4. Rabeb Mizouni, Aziz Salah, Siamak Kolahi, and Rachida Dssouli. *Composition of Use cases using Model Checking and Synchronization* 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems, Springer-Verlag, Paris, France 2006

5. Rabeb Mizouni, Aziz Salah, Siamak Kolahi, and Rachida Dssouli. *Automated Approach for Use Case Composition*, MCSEAI'06: 9th Maghrebian Conference on Information Technologies, Agadir, Morocco, December 2006

6. Rabeb Mizouni, Aziz Salah, Rachida Dssouli, and Siamak Kolahi. *Roles of variables in Use Case Composition.* New Technologies for Distributed Systems (NOTERE'2006), Toulouse, France 2006.

7. Rabeb Mizouni, Aziz Salah, and Rachida Dssouli. *Interaction-Based Scenario Integration.* Workshop on Model Design and Validation, ACM/IEEE Models/UML, Jamaica 2005

8. Aziz Salah, Rabeb Mizouni, Rachida Dssouli, and Benoit Parreaux, *Formal Composition of Distributed Scenario*, International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2004), pp. 213-228, Madrid Spain, September 2004

9. Rabeb Mizouni, Aziz Salah, Rachida Dssouli, and Benoit Parreaux, *Integrating Scenarios with Explicit Loops*, New Technologies for Distributed Systems (NOTERE 2004), Essaidia Morocco, June 2004

10. Aziz Salah, Rabeb Mizouni, and Rachida Dssouli, *Communication Abstraction and Verification in Distributed Scenario Integration*, Workshop Communication Abstraction for Distributed Systems, ECOOP, Oslo Norway, June 2004

1.4 Thesis Outline

This thesis is structured as follows.

In Chapter 2, we present the different notations used nowadays for representing use cases. For each notation, we present its characteristics and their applicability in the system development process. We classify them according to their level of abstraction.

In Chapter 3, we describe the different approaches for composing use cases. For each approach, we describe its advantages and its limitations. We classify them according to their level of automation. A comparison between them according to some criteria is also presented.

In Chapter 4, we present our approach of composing overlapping use cases, where use cases are a variant of extended finite state machines. The variable characterization of the states in different use cases is used as a composition criterion. The steps of the composition are described and illustrated by an example.

In Chapter 5, we describe the explicit composition of the same model enriched with interactions. Interactions describe the semantics of the composition. In this case, use cases do not overlap. A cut and paste operation is performed in order to interpret the different interactions described within the use cases to obtain the overall system automaton. In a second step, implied scenarios are detected and removed by adding additional constraints to the system automaton.

In Chapter 6, we define a novel approach of composing use cases using imperative expressions. In this chapter, we give an overview of the proposed approach, the underlaying assumptions, and its main advantages in requirements analysis phase.

In Chapter 7, we present the formalization of the approach of composition using imperative expressions in the case of finite state automaton. The approach is presented first in the case of composing two use cases in a unique extension point, then in multiple extension points.

In Chapter 8, we present the formalization of of the composition using imperative expressions in the case of extended finite state automaton. The approach is presented first in the case of composing two use cases in a unique extension point, then in multiple extension points. Issues related to variables such as the definition of global variables to the specification vs. the definition of local variables to the use cases are discussed.

In Chapter 9, we present the tool implemented in order to validate our composition approach. A specification of an e-Purchasing system is used to illustrate the approach in each of the proposed use case model.

Finally, in Chapter 10, we summarize the thesis contributions, discuss the obtained results in each of them, and give a list of topics for future research directions.

Chapter 2

Notations

2.1 Introduction

Requirements engineering is the first activity in the life cycle of system development. It consists of an iterative process that encompasses discovering, analyzing, and validating the system requirements. This involves all activities devoted to the identification of user requirements, their analysis in order to derive additional requirements, the documentation of the requirements as a specification, and finally the validation of the documented requirements against user needs. Therefore, requirements engineering is critical for the success of the project. It is the phase where what to build is determined. Any forgotten behaviors, miscommunication with users, and undetected faults lead to considerable changes in the obtained system with high costs.

In the first phase, stakeholders collaborate together to define the user requirements. A requirement is in fact a description of what the system is expected to do. It captures the intended behavior of the system. This behavior may express services, tasks or functions the system is required to perform. According to Young [112], each individual requirement

should be:

- Necessary: If the system can meet prioritized real needs without the requirement, it is not qualified as necessary.
- Feasible: The requirement can be accomplished within reasonable cost and schedule.
- Correct: The facts related to the requirement are accurate and it is technically and legally possible.
- Clear: Written in a not confusing way.
- Concise: The requirement is stated simply with appropriate words in a brief and a succinct way.
- Unambiguous: The requirement can be interpreted in only one way. For this reason, the language used to write the requirement is critical for its understanding. For instance, the natural language is usually misleading while a formal language is uniquely interpreted but requires stakeholder expertise.
- Complete: All conditions under which the requirement applies are stated and it expresses a whole idea or statement. Completeness is hard to achieve and to check too.
- Consistent: Not in conflict with other requirements.
- Verifiable: Implementation of the requirement in the system can be verified.
- Traceable: Can trace back to the source of the requirement and it can be tracked throughout the system (e.g., to the design, code, test,...). It is a very important characteristic that, when verified, helps the maintainability.

- Design independent: Does not impose a specific implementation solution.

- Non-redundant: Not a duplicate requirement.

Usually at the first stage of development, requirements are still not stable and will have to change when their inconsistencies are discovered in the analysis phase. The latter ends up with the derivation of the overall system specification.

The final product of requirements engineering is a requirement document. The format and composition of this document vary from one methodology to another. The requirement document may contain different types of descriptions that vary in their formality and level of details. It may contain a general system description, a list of functional and nonfunctional requirements, a textual description of scenarios, a definition of the different actors, a list of prescribed system components, etc[22]...

A requirements specification is a complete (as much as possible) description of the behavior of the system to be developed. It includes a set of functional requirements that describe all of the interactions that the users will have with the system, and a set of nonfunctional (or supplementary) requirements which impose constraints on its design or implementation (such as performance requirements, design constraints...).

At this level, having formal specifications is particularly useful because they are precise, clear, unambiguous, and may be verifiable. These properties help to discover bugs early in the system lifecycle. The earlier a fault is detected, the cheaper it can be removed. For this reason, formal specification methods can improve productivity and quality.

Finally, validation is concerned with demonstrating that the requirements define the system that the customer really wants. Validation activity is very important because it allows the detection of requirement errors. The properties to check are :

- Validity: Does the system provide the functions which best support the customers needs?
- Consistency: Are there any requirements conflicts because of contradictory constraints?
- Completeness: Are all functions required by the customer included?
- Realism: Can the requirements be implemented given available budget, schedule and technology?
- Verifiability: Can the requirements be checked? Does the delivered system meet the requirements?

Consistency and completeness of the specification have to be established. Usually, completeness is more problematic and hard to validate.

2.1.1 Use Case Definition

Use Cases have proved to be an effective mechanism for the capture of requirements. The notion of use case has been proposed first by Ivar Jacobson [47]. Over the years, many definitions have been given to use cases, let's cite some examples: Jacobson in [47] has defined use cases as :

a narrative document that describes the sequence of events of an actor (an external agent) using a system to complete a process.

Larman in [60] defined them as *stories or cases of using a system. Use cases are not exactly requirements or functional specifications, but they illustrate and imply requirements in the stories they tell.* They represent *narrative descriptions of domain processes in a structure prose format.*

According to Booch [21], a use case is *A description of set of sequences of actions, including variants, that a system performs that yield an observable result of value to an actor.* It is a way to *capture the intended behavior of the system without having to specify how that behavior is implemented...In addition, use cases serve to help validate your architecture and to verify your system as it evolves during development.*

Finally, in a more recent book [48], Ivar Jacobson specified that *A use case models the behavior of a system. A use case is a sequence of actions performed by a system, which yields an observable result that is typically of value for one or more actors or other stakeholders of the system.* During this thesis, we adopt this definition of use case, which we believe to be a more general one.

2.1.2 Why Use Cases?

Use cases are a model to capture software requirements. They facilitate the description of the system requirements by applying a "divide and conquer" strategy, since describing a part of the system is simpler than describing it as a whole. Use case approaches offer several practical advantages :

- Use case modeling (including the writing of use case specifications) is generally regarded as an excellent technique for capturing the functional requirements of a system.
- Use cases are usually easy to describe and to understand by the different stakeholders. They have proved to be easily understandable by business users, making them an excellent bridge between software developers and end users.
- Use cases are design independent. Hence, they avoid early design decisions.

- Use cases help the traceability of the different requirements throughout the design and implementation.

- Use cases are scalable in that the behavior of a large system can be described as a collection of independent but complementary use cases developed incrementally [61].

- Use cases can serve as the basis for estimating, scheduling, and validating effort.

- Use cases are reusable within a project. They can evolve at each iteration from a method of capturing requirements, to development guidelines to programmers, to a test case and finally into user documentation. For instance, test cases (System, User Acceptance and Functional) can also be directly derived from the use cases. As shown in Figure 2, Zielczynski [114] has proposed the pyramid of needs where he defines the traceability of requirements. On the top level are stakeholder needs. Features define supplementary specifications. The further down, the more detailed are the requirement. Use cases describe functional requirements, and supplementary specifications (Supp/Spec) describe non-functional items. In addition, every use case maps to many scenarios, where a scenario is a possible execution of a use case. Scenarios map to test cases in a one to many relationship.

- Use cases are useful for scoping. They make it easy to take a staged delivery approach to projects; they can be relatively easily added and removed from a software project as priorities change.

- Use case specifications can be written in a variety of styles to suit the particular needs of the project. They can be expressed in any language ranged from natural language to formal language.

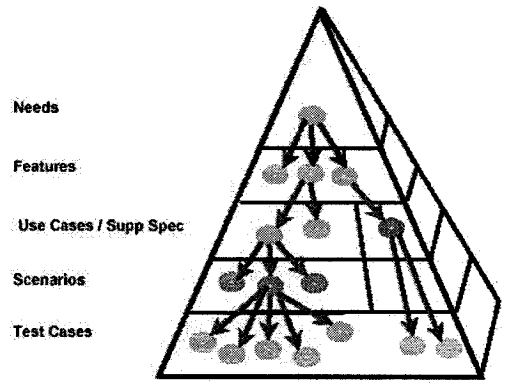


Figure 2: Generating Test Cases from Use Cases [114]

Despite these advantages, use case approaches suffer from a couple of major drawbacks:

- Use cases are often stated in natural languages, lacking formal syntax and semantics. This may make them ambiguous and hard to analyze.
- The dependencies and the relationships among the different use cases are usually hard to analyze. Currently, there is no systematic approaches to determine them [61].
- Inconsistency and incompleteness between use cases are hard to detect before forwarding to design phase.
- Use Cases are often unable to describe non-functional requirements.
- Distribute use cases in order to derive the behavior of the different components is also a challenging task.

2.1.3 Why formal methods?

Many techniques can be used for validating and verifying the user requirements, we cite testing and simulation. Testing has the advantage of having immediate relevance. However, since tests are applied on the product, errors may be detected late and hence the

cost to correct them may be high, in addition to the fact that diagnosis needs complete observability. In contrast, simulation can be performed through design but simulators can be significantly slower than real systems. Both techniques can not be applied exhaustively, and they can only show the presence of bugs, but never prove their absence. Formal methods are complementary techniques to testing and simulation. They emerged to overcome their drawbacks and to allow the exhaustive testing of a user requirements. They provide a set of tools and notations with a formal semantics that allow the unambiguous specification of requirements, their verification against some properties, and the prove of correctness of an implementation with respect to that specification.

Use of formal methods does not a priori guarantee correctness. However, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompletenesses that might otherwise go undetected [28].

Formal methods have been proved to be efficient when the complexity of the system to build is high. They also have a wide acceptance in concurrent systems, distributed systems, and fault tolerant systems. While in the past the use of formal methods in practice seemed hopeless because of the complexity of the models used, success of the application of formal methods in specification and verification have been shown in the last decade [28].

In specification, the use of precise models allows a deep understanding of the system being specified, offering the possibility of discovering design inconsistencies and incompleteness. In verification, two well established approaches are nowadays widely used, which are model checking and theorem proving. They are used to analyze a system for desired properties. Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. It performs an exhaustive exploration

of the state space of the given model. In contrast, theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. Then, a proof has to be conducted, interactively. During this thesis, our main concern is to generate a verifiable specification of the system behavior starting from a set of requirements. Since formal methods help the definition of precise requirements, and they lie on well defined semantics, they provide a framework for developing an automated approach of requirements composition.

2.2 Use Case Notations

Since use case approaches have a wide acceptance in the industry, dozens of notations have emerged in order to define them. Use case notations may describe the set of scenarios as Message Sequence Charts (MSCs) [2], UML Sequence Diagrams (SD) [1], Life Sequence Diagrams (LSCs) [39], state-based notations (such as the behavior section of the Specification and Description Language (SDL), statecharts [37], finite state machines)...

These notations have different levels of abstractions, which make them particularly useful for adding details to an outlined requirements description. In the first stage of requirements engineering, one needs a notation with high level of abstraction since it's mainly used to describe the functionalities of the desired system but not how these functionalities have to be implemented. When forwarding to the implementation level, these notations have to be refined to allow the expression of more details.

The most intuitive use case descriptions are unfortunately the informal ones, where the user represents the desired behavior of the system in his own language. The problem with this kind of representation appears when it comes to interpret the specification since the

natural language may lead to ambiguities. Thus, to benefit from the expressive power given by the natural language and to resolve its constraints, many researchers [96] restrict the considered language to a predefined subset.

Semi-formal or formal language [12, 33] are also widely used for scenario descriptions. The latter can be automated easily but they may be less understandable for stakeholders, while the former have more expressiveness power but difficult, almost impossible, to automate. Using the appropriate notation to serve the desired level of abstraction is always a decision to make when describing use cases. In what follows, we propose a summary of the most used notations in use cases with respect to the classification abstraction and formality (c.f. Figure 3).

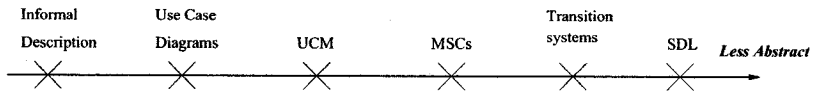


Figure 3: The Abstraction Level of Different Scenario Notations

2.2.1 Abstract Notations

UML Use Case Diagrams

The UML Use Case Diagram [1] is a semi-formal language for system specification. It allows defining an abstract view of the system that models the interactions between the system and its actors. An actor represents a user or another system that will interact in a way or another with the system, while a use case is an external view that represents some actions an actor may perform to achieve a task.

Use Case Diagrams are very useful in the first stage of the software development because they help the description of the requirements in an abstract manner. However, they should

be refined in a latter stage because they omit all details to understand how a task will be performed. Figure 4 shows an example of a UML diagram. It depicts a system where students are enrolling in courses with the potential help of registrars. Professors input the marks students earn on assignments and registrars authorize the distribution of transcripts (report cards) to students.

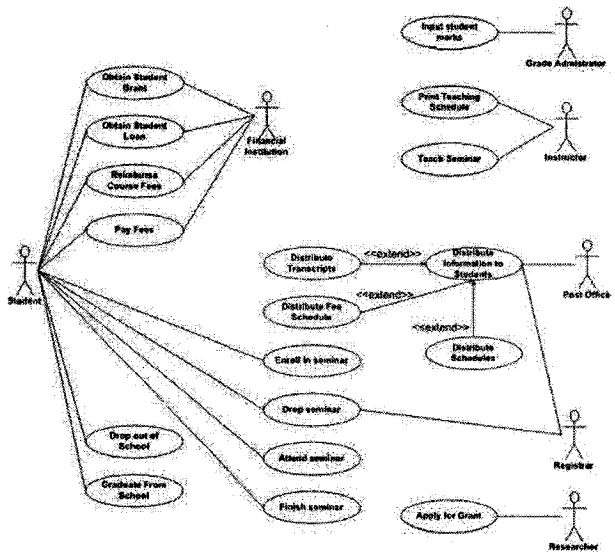


Figure 4: UML Use Case Diagram: Example [3]

UML provides some relationships to describe the dependencies among use cases, we cite `Include`, `Extend`, and `Generalization`. `Include` depicts the fact that one use case includes the behavior of another use case, while `Extend` means that one use case is extended to another use case so that it has the behavior of the former in addition to its own. Extension can only be done at extension points of the use case being extended. While with Include, the inclusion of the behavior is mandatory, with Extend, it is optional. `Generalization` depicts the fact of inheriting parent behavior, adding and overriding with the child's own behavior.

Use Case Maps

Use Case Maps (UCM) [5] are a use case based software engineering technique most useful at the early stages of software development, first introduced by Burh [26]. The notation can be applicable to use case capturing and elicitation, use case validation, as well as high-level architectural design and test case generation. They represent a graphical models that describe the functional requirements and high-level design with causally linked responsibilities [13]. The notation allows representing use cases as sets of causal paths, specifying the way the system can evolve without specifying any message details. UCM are very abstract, which promotes flexibility. Hence, many MSCs can be valid according to a same UCM. For this reason, they are very useful in the first stage of requirement elicitation where details are not yet available.

A UCM scenario starts with a triggering event that corresponds to a precondition and ends with resulting events that represent the post-conditions. The components are determined according to the role they are playing and they are optional. UCM notation has the possibility to express alternative and concurrent sub-scenarios. Moreover, the UCM offer an explicit notation for the scenario integration.

Use Case Navigator (UCMENav) [82] is a tool that edits and analyzes UCMs. UCMNAV is used for creating and navigating UCMs and for storing them as XML file. However, the UMC specification presents several problems, mainly a problem of maintainability because the tool was developed in a ad-hoc manner. In addition, the user interface is not very friendly. Hence, it was replaced by the jUCMNav tool. jUCMNav [100] is an Eclipse plug-in project which uses both the Graphical Editing Framework (GEF) and the Eclipse Modeling Framework (EMF). In addition to editing and analysing UCMs, jUCMNav is augmented to

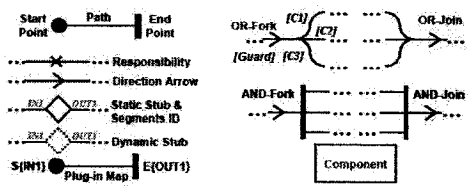


Figure 5: Basic Notations of Use Case Maps [100]

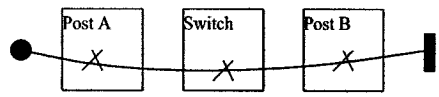


Figure 6: Example of Use Case Maps

produce scenario files in XML and to generate MSCs out of them [12]. In [49], jUCMNav has been linked to Telelogic DOORS [4], which is a requirements management system that supports various types of requirements objects, attribute types, and traceability links. The tool allows UCM scenario models to be imported in DOORS and linked to other types of requirements.

Message Sequence Charts MSCs

Message Sequence Chart (MSC or basic MSC, bMSC) is one of the most popular and used scenario specification languages that describes the interaction between a number of independent message passing instances. It allows both graphical and formal representation of scenarios, thing that makes it very practical. It supports complete and partial specification and it could be used to express both desired and forbidden behavior. A message can be as simple as a signal and as complex as a data packet. Two events are associated to messages: send and receive. Timers can be inserted to model time constraints. In addition, this language is widely applicable to various application domains [2]. The MSC language allows formulating compositions within the MSC by mean of *inline expressions*. They can express weak sequential composition, alternative composition, optional composition, parallel composition and finally loops with bounded number of iterations. An MSC example of a timed

communication between an Entity A and an Entity B is shown in Figure 7.

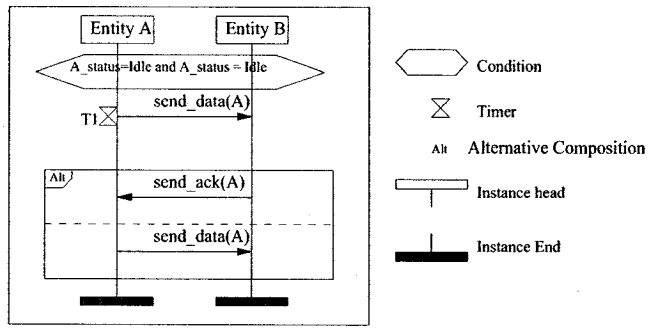


Figure 7: MSC Specification

Many trials aimed at extending the semantics of MSCs. Zheng and Khendek [113] extended the timed MSCs with instance delay in order to enhance its expressiveness. Sengupta and Cleaveland [94] extended MSCs to express conditional scenarios by differentiating between triggers and actions, and between "extensible" and "complete" partial scenarios.

UML Sequence Diagrams (SDs)

UML sequence diagrams are similar to the MSCs notation. UML sequence diagrams are commonly used for both analysis and design purposes. They represent one of the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within the system.

One of the primary uses of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement. Use cases are often refined into one or more sequence diagrams.

UML sequence diagrams focuses less on messages themselves and more on the order in which messages occur. Nevertheless, most sequence diagrams will describe what messages are sent between system's objects as well as the order in which they occur. The messages sent

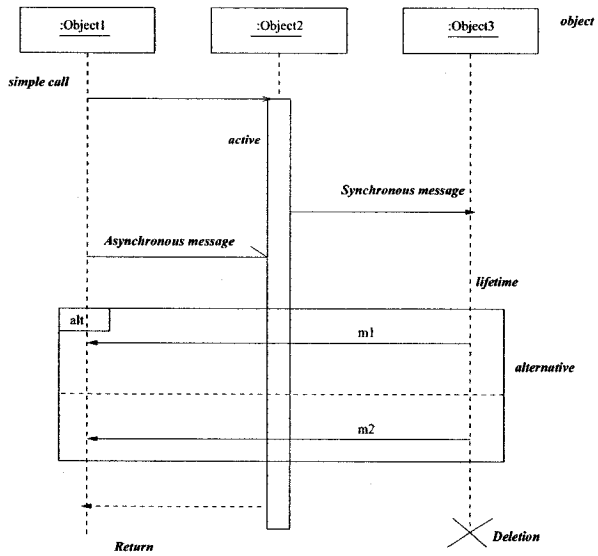
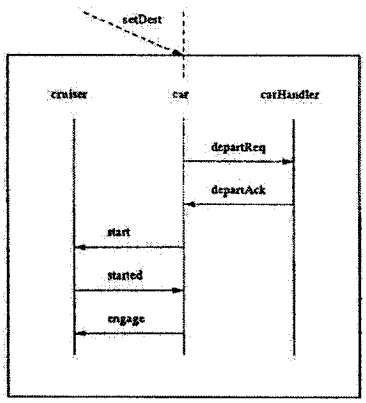


Figure 8: Notions of Sequence Diagrams

and received can be synchronous (a stick arrowhead) or asynchronous (a solid arrowhead). SDs allow the specification of alternatives, loops and options. Alternatives are used to designate a mutually exclusive choice between two or more message sequences. Options are used to model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur. Loops model a repetitive sequence. Figure 8 shows an example of a sequence diagram.

LSCs: Live Sequence Charts

LSCs has been proposed as an extension of message sequence diagrams [29]. They extend MSC by the notion of liveness. LSC distinguishes between the possible and the necessary behavior by the introduction of universal chart, where the chart should be executed in all the runs of the system, and existential chart, where it should be executed at least in a run. Forbidden scenarios can be specified. Figure 9 shows an example of a universal chart. It describes a car departing from a terminal. The instances participating in this use case



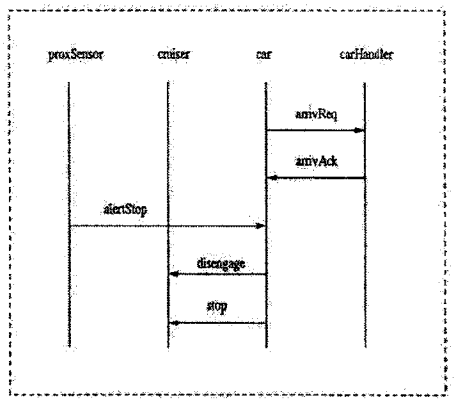


Figure 9: Example of Universal LSCs [29] Figure 10: Example of Existential LSCs [29]

are *cruiser*, *car*, and *carHandler*. *SetDest* is a simple kind of activation condition (an activation message).

Figure 10 shows an example of an existential chart, depicted by dashed borderlines. The chart describes a possible stopping of a car at a terminal. This chart has to be satisfied in at least one run. LSCs do not have a high level view like HMSCs, but Bontemps *et al.* [20] have extended the notation with composition operators and defined their semantics in terms of ω-regular traces.

2.2.2 State Based Notations

Petri Nets (PN)

Petri Nets [85] are formal, graphical, executable techniques for the specification and analysis of concurrent, discrete-event dynamic systems. A Petri net is assembled from places and transitions. Places, depicted as circles or ellipses, represent resources that can be available or not. The availability of a resource is shown by a black dot inside the circle. When

resources are available, we also say that the place is marked. Individual resources are abstractly referred to as tokens. Transitions, depicted by rectangles or squares, are the active elements of a net. A transition that occurs (or fires) can remove tokens from some places and insert tokens into other places. In order to denote the tokens that are moved by a transition, arrows, so-called edges, are drawn from places to transitions and from transitions to places. Each arrow can be annotated by a number that indicates the number of tokens that are moved by this edge. Formally, a Petri Net is defined as follows [81]:

Definition 1. *A Petri net is a 5-tuple* (S, T, F, M_0, W)*, where:*

- S *is a set of places.*
- T *is a set of transitions.*
- F *is a set of edges known as a flow relation. The set* F *is subject to the constraint that no edge may connect two places or two transitions, or more formally:* $F \subseteq (S \times T) \cup (T \times S)$.
- $M_0 : S \to \mathbb{N}$ *is an initial marking, where for each place* $s \in S$*, there are* $n_s \in \mathbb{N}$ *tokens.*
- $W : F \to \mathbb{N}^+$ *is a set of edge weights, which assigns to each edge* $f \in F$ *some* $n \in \mathbb{N}^+$ *denoting how many tokens are consumed from a place by a transition, or alternatively, how many tokens are produced by a transition and put into each place.*

PNs are used to capture sequential, alternative and concurrent scenarios. Like UCMs, the ordering of actions in PNs is based on the causality. Efforts are being made to extend the PNs with time considerations.

Finite State Machines

Finite state machines (FSM) have become a standard model for representing system behavior. Numerous specification notations are based on the concept of FSMs. UML and SDL incorporate FSM notations. An FSM is defined by the following: a finite set of states, a finite set of events, and a set of transitions that map some state-event pairs to other states. Actions are normally associated with transitions. Since the level of FSM specifications is pretty low, Harel introduced hierarchical statecharts in order to reduce the size of the specifications. State machines can be constructed from regular expressions. More formally,

Definition 2. *An FSM is a 5-tuple* $M = (Q, \Sigma, q_0, \delta, F)$

- Q *is the finite set of states.*
- Σ *is the set of input events.*
- $\delta : Q \times \Sigma \rightarrow Q$ *is the transition relation.*
- $q_0 \in Q$ *is the initial state.*
- $F \subset Q$ *is the final or accepting states.*

Transition relation are defined according to the current state, the input that allows the firing of the transition, and the next state of the machine.

Extended FSM (EFSM) model describes a module process as an FSM extended with variables. In an extended finite state machine model, the transition can be expressed by an "if statement" and associated with a set of enabling conditions. When all of the conditions are satisfied, the transition is fired. It brings the machine from the current state to the next state and and may update the values of the variables by performing the specified data operations [30].

Statecharts

Statecharts are finite state machines that model the behavior of systems. They were introduced first by Harel [37] and have a widespread usage since a variant has become part of UML. They extend the conventional state transition diagrams with hierarchy, concurrency, and communication. The first notion is obtained by embedding one statechart in the state of another statechart. While the second notion, called *orthogonality* by Harel, is obtained by parallel statechart composition. Statechart are state-event driven diagrams. To capture system behavior, each scenario will be represented by a closed statechart having a single initial state and a single final one.

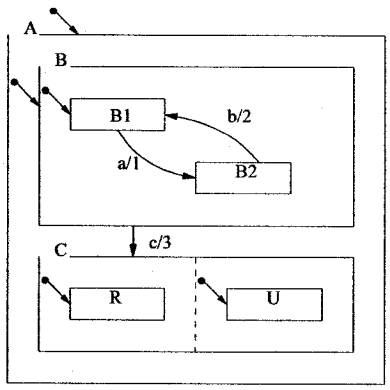


Figure 11: Example of a Statechart

The advantage of using statechart in use case description and analysis is mainly the hierarchy they are offering because in the case of other plain state automaton, the state explosion problem raises when the specification deals with parallel scenarios.

Labeled Transition System : LTS

LTS [50] is very simple specification formalism for describing the behavior of a system component. It is used to represent the behavior of a component, which has a finite number

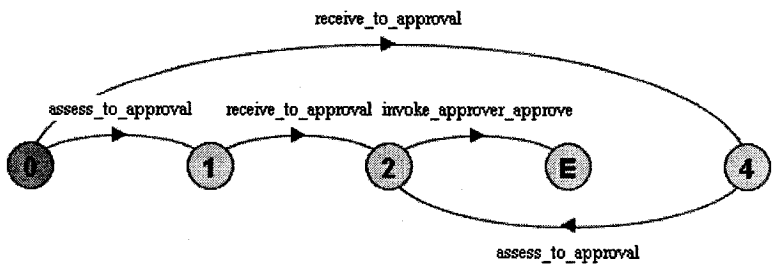


Figure 12: Example of an LTS [66]

of possible values for its state and a finite number of operations. No operation parameters are considered; only the order of the operations is specified. An LTS is represented by a state diagram, where transitions are labeled by the operation. An LTS may be either deterministic or non deterministic. Formally, an LTS is:

Definition 3. *An LTS is a tuple* (S, L, Δ, s_0) *where:*

- S *is the set of states*
- L *is the set of labels;* $L = \alpha(P)$ *where* $\alpha(P)$ *is the set of communicating alphabet of* P.
- $\Delta : S \times L \rightarrow S$ *is the transition function*
- s_0 *is the initial state.*

The semantics of LOTOS (Language Of Temporal Ordering Specifications) [19] for example, one of the formal description techniques, is based on the LTS model. Figure 12 depicts an example of an LTS.

Specification Description Language: SDL

SDL [17] is a formal specification language widely used in telecommunications. SDL covers different levels of abstraction from a broad overview down to detailed design level. SDL

language is used to describe the structural and architectural properties of the system as well as its behavioral properties. Hence, an SDL specification covers three aspects:

- The structure of the system in terms of processes and interconnections.
- The dynamic behavior of each process, its interactions with the other processes and with the environment.
- The data structures manipulated by the processes.

The two first aspects of the system are architecture oriented (c.f. Figure 13). The communication between processes is asynchronous, with a FIFO queues. The semantics of the

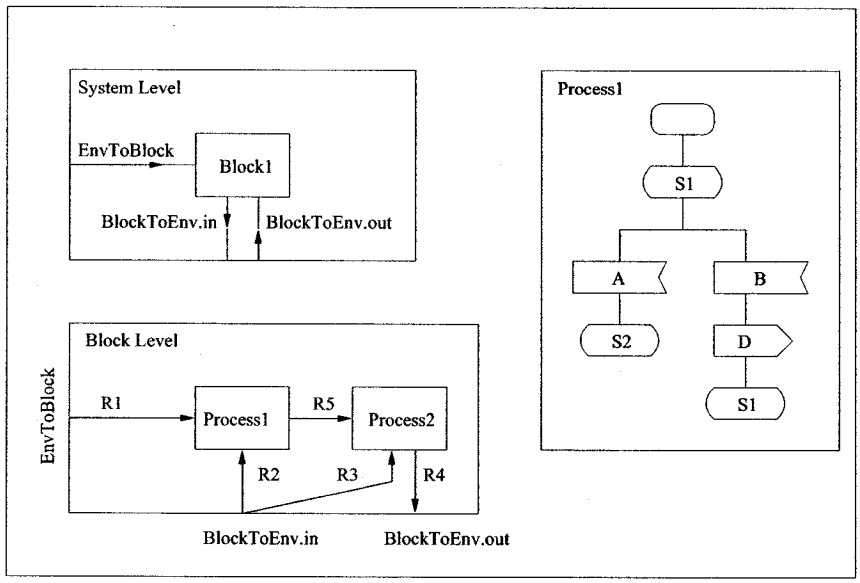


Figure 13: SDL Specification

language is based on extended finite state automata that have been augmented with features for specifying abstract data types.

2.2.3 Use Case Composition Notations

In order to compose use cases explicitly, some notations have been developed to specify the relationships between the different use cases.

HMSCs : High-Level MSCs

One of the techniques to compose basic MSCs is the High-Level MSCs (HMSCs). They provide a graphical representation to combine MSCs together in order to describe alternative, sequential, iterating, and non-deterministic execution of scenarios in an attractive graphical layout. *An HMSC consists of a collection of components, enclosed by a frame. The components are thought of as complex MSCs that operate in parallel. Every component consists of a number of nodes and a number of arrows that imply an order on the nodes* [67]. In Figure 14, we show an example of HMSC.

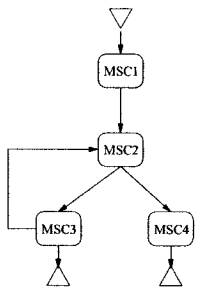


Figure 14: HMSC Specification

The HMSCs are facing the problem of process divergence and the non-local branching choice. These problems were tackled by Ben-Abdallah *et al.* [36]. It is mainly due to the asynchronous interpretations of HMSCs. The first problem depicts the case when a process is sending messages to another, and the latter is not consuming them. Figure 15 (a) shows such problem. The second problem happens when there is an alternative choice between

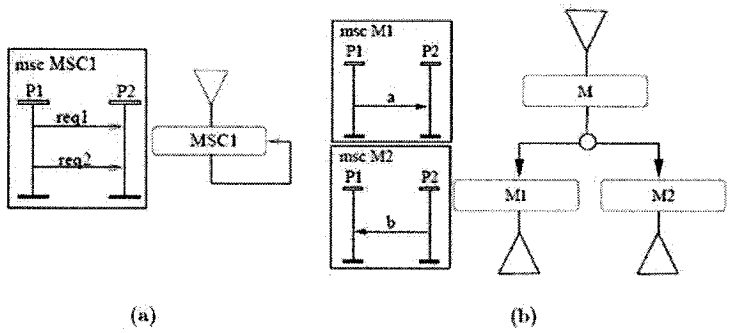


Figure 15: Example of Process Divergence and Non-Local Choice in HMSC [36]

two MCSs. The system may not evolve in the same MCSs because the first events in all the MSCs (of the branching point) are not sent by the same process. Figure 15 (b) depicts such problem. We notice that These problems raise the problem of HMSC realizability. Realizability depicts the existence of an implementation that exhibits the same behavior of the HMSC.

Dependency Chart

Since the current integration methods have limited support for abstraction and decomposition, the authors proposed in [88] a new diagram called dependency chart. It is a notation that :

- allows a clear understanding of the system high-level dependencies and connections between scenarios.
- facilitates hierarchical decomposition.
- helps in scenario reuse and testing.

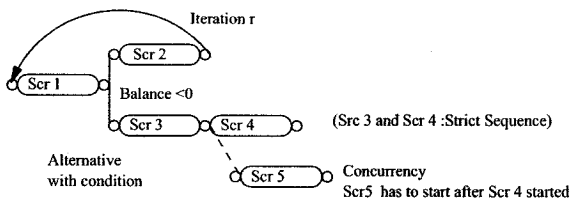


Figure 16: Dependency Chart

The notation they are proposing offers to express abstraction, causal, and temporal dependencies (Figure 16). The dependency charts depict mainly the static relations between scenarios.

2.3 Summary

In this chapter, we presented the most used notations in use case based approaches. These notations differ in their level of abstraction, expressiveness and formality level. The choice of one of these notations depends mainly on the stage of the lifecycle where the notation is used, the formality level of the composition approach, and the automation level of the composition approach. In the next chapter, we will overview the different approaches of use case composition in the state of the art.

Chapter 3

State of the Art

3.1 Introduction

Use Cases are usually used in order to derive formal specification of the intended system that is conforming to the requirements of the user. In all cases, same questions have to be answered:

1. How to represent use cases? As explained before, many notations with different levels of abstraction are currently used.
2. How to compose use cases? Based on what criteria use cases will be integrated?
3. What is the target model the composition algorithms will derive?
4. What is the level of automation of the synthesis approach?
5. How to validate the overall specification? Decisions on the previous questions will affect the approach undertaken for validation.

Many approaches have been developed during the last decade to compose use cases and generate, usually, a state-based model. State-based models are needed to validate and verify the user requirements in order to detect problems as soon as possible. These approaches are different in many aspects:

- The identification of the system state is usually done on a trace [39, 57, 58] or variable identification [97, 104, 109] bases. In the latter approaches, a variable state vector is defined and it is used in the generation of state based models.
- Often, the generated state model is either a statechart [109, 32, 58, 68] or an automaton [97, 38].
- Some researches have been conducted to improve the model used by timing aspects [92, 31].

In the next sections, we will overview a set of these approaches. We categorize them to two kinds: approaches that target the improvements of the notations used to express use cases, and approaches that target the improvements of the composition mechanism itself. These approaches are classified according to their level of automation.

3.2 Notation Improvement

3.2.1 Non-automated Approaches

Glinz [32] uses statecharts to formally model use cases. His objective is to integrate different use cases with respect to two conditions. The first one is that the resulting model shows the relationship between the use cases and keeps their internal structure unchanged. The second condition is to detect inconsistencies between use cases.

The proposed approach allows only the composition of disjoint use cases. When it comes to overlapping ones, the author proposes either to decompose them or to use them such that we obtain a single disjoint use case. The author defined four constructors to build a use case from elementary ones: the sequencing constructor, the alternative constructor, the iteration constructor and the concurrency constructor.

The method has the advantage of detecting deadlocks when mapping the generated statechart of the composition to a flat one. However, when it comes to use this approach in a practical way, the resulting statechart from the integration of many use cases becomes unreadable, reason for what in the ADORA project [34], they used the Jackson-style diagrams [111]. Glinz is presenting a notation, rather than a methodology, that can clarify the relationships between use cases.

As an extension of Glinz's work, Ryser [88] introduces a new kind of charts and notations to model dependencies among use cases. The motivation of this work is that usually a single use case models an aspect of the system. A system is modeled by more than one scenario. So, it is useful to have a notation that captures clearly these inter-scenarios dependencies.

This work is closely related to the one of Glinz [32], however it is more general since it offers more operators. It also provides more dependencies than the UML use case relationships (generalization, Include, Extend).

Hsia *et al.* [46] proposed an approach where they start from scenarios elicitation to reach scenarios verification. In the first phase, the analyst constructs a *scenario tree* according to a user view. Scenarios consist of a set of events that change the system state, trigger another events or do both. At the end of this phase, the user has a scenario schema.

The second phase takes care of scenario formalization where the scenario tree is converted into grammar and the latter is used to construct the conceptual state machines. Both the grammar and the conceptual state machine represent the abstract formal model of the system behavior. In the third phase, the abstract model is verified against inconsistencies, redundancies, and incompleteness. In the next phase, the verified formal model is used to generate automatically scenarios, from which a prototype is generated to construct the expected system. Finally, the prototype is used to validate scenarios and demonstrate their validity. The advantage of this method is the fact of ending up with complete, deterministic and consistent scenarios when seen individually. However, there is no guarantee that problems of inconsistency between scenarios and undesired behavior will not arise when integrating these scenarios.

Bordeleau ***et al.*** [23, 24] propose to use UCMs at the requirements definition level in order to derive from them detailed interaction diagrams, and to proceed from them hierarchical state machines for each component using patterns. They define two steps for the generation of hierarchical state machines. The first one is to map a role to the behavior of an object in a particular scenario (for all the scenarios they have), to which they design a state machine. The second step consists of integrating the different roles of the component existing (in all the scenarios they are dealing with) on a hierarchical state machine with respect to the temporal and the logical ordering of the roles. They propose a catalog for integration patterns for specific development contexts. The pattern takes into account temporal and logical relationship when integrating a scenario (a role) with the already handled roles of components. The catalog provides the patterns for both Scenario Partitioning and

State Machine Integration. The challenging part of this approach is how to recognize patterns and how to combine their semantics.

Harel [38] argues the possibility of generating code from requirements. The approach consists of *playing-in* and *playing-out* the requirements using as input LSCs [29]. The *Play-in* concept is high-level user-friendly graphical interface (GUI) that allows the user to capture the requirements of the target system or an abstract version thereof. It consists of describing the sequencing of the model messages as desired by the user. The second concept, play-out, consists of executing the requirements interactively, without building the model or writing the code. During this phase, the role of the user will be limited to an "end-user" and environment actions only.

Using this model, a Play-in/Play-out Engine is implemented. During the *Play-in*, the *Play engine* will generate automatically the LSCs of the user requirements entered interactively using the GUI. During the *Play-out* [40], the user will play the GUI application as he would have done when simulating a system model. A BDD model checker is used to verify that, always, at least one of the universal chart is active. The tool translates the play out task into the corresponding model, runs the model checker and then injects the obtained counterexample into the play engine. Since the model checker they use are BDD-based, the variable ordering is crucial to the verification task. Currently, they don't generate the order automatically; instead they use structural information from the LSC specification in order to derive a good variable ordering. The tool can specify asynchronous models. However, the LSC semantics requires that objects be synchronized while iterating during loops. In

the verification of the play-out scenarios, the model checker is used just to verify the action done by the user and the consequences it produces. They end up with no exhaustive verification. Hence, some implied scenarios can remain undetected.

Somé *et al.* [98, 97] proposed an algorithm towards the synthesis of parametric timed automata [9] from structured textual scenarios or extended MSCs. In order to derive the desired behavior, they presented an incremental algorithm that merges scenarios with respect to timing constraints. In this approach, each state is characterized by the condition that holds in it expressed in terms of variable valuations. The first state of each partial run over a use case includes the use case pre-condition. Hence, when merging a new use case, an algorithm first checks the existence of such state. In the case of positive answer, the algorithm, starts to insert transitions while in the opposite situation, omissions in the original scenarios are detected.

Solving these problems needs the user assistance. The algorithm has the advantage of preserving the temporal constraints associated with the scenarios, which is seldom the case of other semi-automated synthesis techniques. This work was extended [92] with improved support for automation. The authors explored the implicit integration of scenarios rather than the explicit one. An algorithm for optimizing the obtained timed automata has been also developed.

Giese [31] also presented an approach towards the synthesis of parametric timed automata from scenarios. UN-timed scenarios are first derived according to existing approaches like the one of Uchitel et al. [104], then the timing constraints are added in

an incremental manner as time boundaries. The approach detects all the timing conflicts that can occur when integrating different scenarios, and hence can be adjusted. Contrarily to the approach in [97], Giese is synthesizing more than one automaton at the same time.

3.2.2 Automated Approaches

Robert *et al.* in [86] have proposed a tool for linking MSCs to SDL specification, MSC2SDL. Starting from a set of MSCs and an SDL architectural model, an SDL specification is derived. During the SDL synthesis, the compatibility of the MSCs architectural commitments and the SDL architecture has to be verified. Later on, Khendek *et al.* [52] proposed an approach for the incremental construction of SDL specification from an existing SDL model and a set of newly specified MSCs. An extension of this work has been presented in [25] in order to translate MSC and UML specifications automatically into a full SDL specification. UML is used to specify the architecture of the system while MSCs are used to give the different scenarios.

Amyot *et al.* [13] proposed an approach where the designer focuses on the main functional aspects of the system to be specified. This means that scenarios are described in terms of causality and responsibilities, allowing a higher-level of abstraction then the diagrams where the message sequence exchange is specified. They utilize Use cases Maps (UCM) for the definition of different scenarios, leading to a formal specification of the system from where they can generate functional tests. Hence, a scenario is seen as a sequence of responsibilities (events and activities) that occur internally or externally, and that are regrouped together, in a causal manner, to serve certain functionality. The advantage of

this view is that they do not focus on architectural issues early in the process. However, they do not give any idea about the verification of the completeness and the consistency of the scenarios that are provided.

Recently, they presented a work that aims to automatically generate MSCs from UCM specification [12]. They presented a two-step generation process: the first step extracts individual scenarios from UCMs and store them as XML files, while the second step consists of transforming the latter to MSCs using XSLT (eXtensible Stylesheet Language Transformation). Their approach decouples the traversal of UCMs from the generation of the target scenarios. The output is a collection of scenarios where sequences and concurrency are preserved if all the UCMs are well nested [12]. Finally, a work is under progress [42] to derive an SDL specification from UCMs. The idea is to derive from UCMs MSCs and HMSCs using the UCM navigator tool, UCMNav, and to adopt the output to be submitted to MSC2SDL tool [25].

3.3 Integration Improvement

3.3.1 Non-Automated Approaches

Alur *et al.* [10] presented an algorithm that checks the completeness and detects unspecified scenarios that are implied from the combination of different finite MSCs. In their work, they define two kinds of realizability: safe and weak. An MSC is realizable if there exist concurrent automata which implement precisely the MSC. If the obtained automata are deadlock free, the MSC is said to be safe realizable, else it is called weak realizable. In the second case, the algorithm they are proposing produces missing implied (partial) scenarios to help guide the designer in refining and extending the specification. Moreover,

they have applied this approach on different communication architectures (asynchronous and synchronous ones, FIFO, Non FIFO, message buffering...).

In addition, the developed algorithm refers to the designer when an implied MSC is detected, and it generates automatically the state machines from MSCs in the case of safe realizability. They also come up with a formal verification of the algorithm that they are using. However, the proposed approach is only applicable to finite state machines, which is not always the case in the practical word. In the same range of idea, the decidability and the realizability of HMSCs have been studied [64, 11, 80]. It has been proven that synchronous HMSC are decidable while asynchronous ones are decidable only under some restrictions.

Mäkinen ***et al.*** [68] presented an algorithm (MAS : Minimally Adequate Synthesizer) which synthesizes UML statechart diagrams from sequence diagrams in an interactive manner. The algorithm interacts with the user to guide the process in the critical points in order to eliminate the undesirable generalization. In addition, it detects inaccurate or incomplete sequence diagrams. Hence, the designer plays the role of the teacher and the algorithm plays the one of the learner. When the designer rejects a query, he gives a counterexample. A counterexample could be positive, means additional information are given to the algorithm, or negative, which means the corresponding path is forbidden. The methodology used for the generation of the statechart consists of transforming the sequence diagrams describing the scenarios on *traces of string.* The latter are transformed to an observation table where all the traces of the sequence diagram, inferring to states in the automaton, are described and compared according to some criteria. This algorithm ends up with an

automaton accepting the desired language.

Muccini, in [78], proposed an algorithm to detect implied use cases without construction of the Labeled Transition System (LTS) model, contrarily to [101]. Non-construction of the LTS model will save time and prevent from state space explosion. In addition, the designer will not need to put in parallel the synthesized components LTS. He proposed an algorithm to identify implied scenarios in specification composed by MSC and HMSC notations. The approach is based on the enriching of possible but not specified behaviors. Therefore, for each pair of components in an enriched node, if they are communicating using enriched events, an implied scenario is detected.

When the author applied this algorithm to the example in [103], he detected more implied scenarios, which can be considered as a positive point for the proposed approach. However, the author did not yet verify the completeness and the correctness of the algorithm. Formal description of the algorithm has to be provided. As an extension of this work, the author proposed in [79] to detect implied behaviors from the synthesis of non-local branching choices in HMSC graphs.

3.3.2 Automated Approaches

Koskimies *et al.* [57] proposed a synthesis algorithm that integrates scenario diagrams. The algorithm takes as input extended MSCs, and generates automatically OMT (Object Modeling Techniques) statecharts as output. In the SCED [56] tool they have proposed, the FSMs are generated from traces by detection of identical states in a way that makes the final output FSM deterministic. However, this is a very restrictive condition. Since

no informational semantics are used in this detection, errors of identification may occur. Both works [56, 68] were improved by providing a compression algorithm [99] that reduces automatically the size of the statechart by decreasing the number of states and transitions in the diagrams wihle preserving their semantics. However, they don't guarantee that the algorithm they are using for the composite state generation produces the optimal solution.

Uchitel ***et al.*** in [103] and later in [101] proposed to use MSC and OCL to describe the system specification. They presented a tool that builds a labeled transition systems (LTS) behavior model describing the implementation of HMSCs specification. They presented in addition an algorithm that detects implied scenarios. They have integrated these procedures into the Labeled Transition System Analyzer, which allows the model checking and the simulation of the behavior model.

In a later work [104], the appraoch has been extended to capture implied scenarios using partial labeled transition system. They define an MSC language with sound abstract semantics in terms of labeled transitions systems and parallel composition. This language contains the assumptions on how to integrate scenarios explicitly by using HMSCs. They use state labels to add specific domain information, making both the integration and the split of scenarios easier for the stakeholders. Labeled Transition System Analyzer (LTSA) is used to verify the specification against deadlock, safety, and liveness properties. However, the numbers of states will grow exponentially with respect to the number of components, which makes the method vulnerable to the well-known state space explosion problem.

Whittle *et al.* [109] presented an algorithm 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 work stresses the importance of obtaining a readable, well understood and modifiable specification. They use state characterization as a composition criterion. In fact, the evaluation of states is used to detect loops within the same instance and to merge states within different SDs. When conflict is detected, the error is reported to the designer and the algorithm should start again.

The generated statecharts benefit from the hierarchy and the orthogonality properties, however it can be non-deterministic. It is true that the algorithm they propose reduces the number of states however no clear semantics are defined for that. As a result, explicit knowledge may be lost within the synthesis algorithm producing misleading synthesis results. In a more recent work [110, 108] , the authors presented a case study that synthesizes UML statecharts from scenarios of the Air Traffic Control domain. The study consists of comparing the synthesized statecharts and the existing manually developed ones for evaluation. They applied the algorithm in [109] to the weather control logic subsystem of CTAS (Center TRASCON Automation System) which is responsible for advising the other subsystem of the weather forecast updates.

3.4 Summary

Table 1 presents a comparison of the different approaches. It summarizes the source notations of each of the approaches, the used criteria for composing the different use cases,

Authors	Initial Use Case Notations	Type of Composition	Synthesized Model	Automation Degree	Incremental	Tool Support	Implied Scenarios Detection
Bordeleau et al. [23]	MSC	HMSCs	Automaton	Manual	Yes	No	N/A
Uchitel et al. [105]	MSCs	HMSCs	LTS	Full	No	LTSA-MSC [101]	Yes [104]
Harel et al. [39]	LSC	Implicit	Automaton SStatecharts	Manual	No	Yes	Yes
Alur et al. [10]	MSCs	Implicit	CFSMs [a]	Manual	No	No	Yes
Mäkinen and Systä	SD	implicit	Statecharts	Semi	No	MAS	Yes
Leue et al.[62]	MSCs	HMSCs	ROOM[b]	Full	No	MESA [18]	Yes[c]
Whittle et al. [109]	SD	Implicit	Statecharts	Full	No	Yes	Condition Conflict
Amyot et al. [12] (SPEC-VALUE)	UCMs	No	Lotos [65][d]	Manual	No	No	No[e]
Salah et al. [91]	Tree	Operators	Timed Automata	Full	No	SCENA	No
Glinz [32]	Statecharts	Operators	Statecharts	Manual	No	No	Yes [f]
Ryser [88]	Statecharts	Dependency Charts	Statecharts	Manual	No	No	N/A
Somé et al. [97]	Text	Operators	Timed Automaton	Manual	Yes	No	Temporal Conflicts
Koskimies et al. [56]	Trace Diagrams	Implicit	State Machine	Full	No	SCED[93]	No
Hsia *et al.* [46]	Tree	implicit	State Model	Semi	No	Prototype	No [g]
Giese [31]	LTS	implicit/ explicit	Timed Automata	Full	Yes	No	Temporal Conflict

[a] Communicating Finite State Machines
[b] Real-time Object Oriented Modeling
[c] Non-local choice, Process divergence, and timing inconsistencies
[d] Language of Temporal Ordering Specifications
[e] Verification is performed in order to detect undesired behaviors
[f] The approach is able to detect unconsistencies between use cases
[g] Only individual scenarios are verified against inconsistencies, completeness, and redundancies.

Table 1: Composition approaches comparison

the generated notations, the automation level, the incremental aspect of the approach, the tool implemented to validate the approach, and finally the generation or not of implied scenarios. We draw the attention to some limitations:

- The models used to define use cases do not support variables when the approach has a support for composition. Timing constraints [97, 92, 31] or conditions on use case execution [109] may be added.
- All the presented approaches play with use cases as blocks if they are performing an explicit composition.
- Most of the approaches are not incremental (from the constructed state model, we cannot add another use case and construct a new state model). However, the requirements of the user in an early stage of the development process are incremental because they are usually prone to changes.
- Implied scenarios are hard to detect and remove in most of the cases, especially if the type of composition used is the implicit one.

Chapter 4

Implicit Composition of Use Cases with Variable-based State Characterization

4.1 Introduction

Requirements engineering encompasses activities ranging from requirements analysis and elicitation to specification and validation. Even a single activity such as requirements elicitation, is likely to deploy multiple participants who will have multiple perspectives.

Models play a key role in many aspects of requirements analysis and design. Developers build models of the problem domain to understand the relationships between stakeholders and their goals and build models of the system under development to reason about its structure, behavior, and function [90].

[1]Published in [73] and presented in [74]

When describing the system functionalities, the stakeholders describe a set of actions, usually related by a sequencing order. The set of actions represent in fact a system use case. Each execution of the use case is a system trace. When specifying real application using such modeling approach, it is hard to keep the use cases independent. Usually, this activity will end up with the obtention of a set of overlapping use cases.

Obtaining overlapping use cases may have two different causes:

- It may show the fact that the two described functionalities of the system share a common behavior. A classical example of this case is the activity of withdrawing or depositing money in an ATM (Automatic Teller Machine). The client will have to insert his card, pass an identification step, and then collect his card after finishing the operation he asked for (withdraw or deposit money).

- It may denote the fact that the two use cases are different views of the same requirement. For complex systems, use cases are usually constructed and manipulated by distributed teams, each working on a partial view of the overall system. When regrouping these use cases to construct the overall system behavior, such views have to be merged and integrated in a consistent one.

In order to merge a set of possible overlapping use cases, a composition approach that detects the common behavior and composes according to that behavior has to be provided. In such approach, use cases are not composed as blocks. A search in the internal structure of the use cases and a detection of the places where to merge have to be processed. Consequently, a criterion of composition has to be defined.

To achieve this goal, we present a language to describe use cases by sequences of actions.

We assume a scenario to be an execution of a use case. Our model is based on a variable-based state characterization where we distinguish between two kinds of variables: control and location variables. Control variables are used to define predicates that guard the transitions, while location variables are used to characterize the locations of the automaton. Location variables have the advantage of reducing the locations number of the automaton. The state variable characterization is our criterion for merging. In other words, two states are merged (either belonging to different use cases or to the same one) if they have the same state characterization.

From the textual description, a use case graph is constructed. The latter passes through several transformations before obtaining the use case automaton. The designer can define textually or graphically the set of use cases. A template of the textual specification is given in order to automate the process of generating an overall specification. If the designer enters a textual description, a parsing operation is achieved in oder to generate the graph representing this textual expression. The overall behavior of the system is obtained by integrating different use case graphs.

This chapter is organized as follows.

- Section 4.2 presents the use case textual and graphical descriptions.
- Section 4.3 describes the approach we are proposing to transform the use case graph.
- Section 4.4 assembles the puzzle and shows how to obtain an automaton from use cases.
- Section 4.5 presents the Alternating Bit Protocol as an example to illustrate our approach.

4.2 Use Case Description

A use case describes a part of system behavior. It is composed of different scenarios. Each one is represented by a sequence of actions that meets the requirement of the functionalities the user specifies.

Our approach aims at defining a formal but friendly representation of use cases. So, a use case is described with a set of actions. Each action describes the system state evolution. To preserve the causality between the actions, their sequencing is also explicitly indicated.

4.2.1 Preliminaries

The modelization of systems requires the definition of the set of labels and a set of discrete state variables. The labels are representing the events on which the objects of the distributed system synchronize. In practice, state variables have symbolic names. However, we will use here a vector-based notation because it is more convenient to present the general case. The state of the system is represented by a state vector $V = (v_1, v_2, .., v_k)$ where v_i is the value of state variable $V[i]$ and k is the number of state variables.

The execution of each action will be associated with state variable conditions. These conditions express constraints on the variable values that, if they are satisfied, they allow the execution of the action. We call them partial pre-condition and partial post-condition. Those conditions are qualified to be partial because they have to be completed by the fact that this action takes place before and after specific actions in the use case.

Contrarily to previous work [92], we differentiate here between two kinds of state variables: the control and the location variables. Consequently, we split the state vector V into two components which we call control vector V_{ctrl} and location vector V_{loc}. How to decide

whether a state variable will belong to the control vector or the location vector is the task of the analyst. As mentioned earlier, a use case is described by a set of actions. Let's give the definition of actions:

Definition 4. *An action a is a structure* $a = (ppre\_loc, ppre\_ctrl, lab, ppost\_loc, ppost\_ctrl)$ *where:*

- *a.ppre_loc and a.ppost_loc are partial pre and post conditions on location vector variables,*
- *a.lab is a label representing a synchronization event*
- *a.ppre_ctrl and a.ppost_ctrl are partial pre and post a condition on control vector variables*

An action is enabled by the action preceding it in the use case. To execute action a, the state vector should fulfill both *a.ppre_loc* and *a.ppre_ctrl*. At that moment, the event *a.lab* is observed. Afterward, the system moves into a state which verifies both *a.ppost_loc* and *a.ppost_ctrl*.

The syntax of *a.ppre_loc* and *a.ppost_loc* is defined by a conjunction of elementary constraints in the form $(V[i] = v)$ where v is a constant in $dom(V[i])$. However, *a.ppre_ctrl* accepts a more expressive syntax where elementary constraints are $(V[i]\#c)$ assuming that $\#$ denotes a binary relations. In contrast, *a.ppost_ctrl* is defined " à la Z" and constraints the relation between the control vector state V_{ctrl} before the execution of a and V'_{ctrl} the state afterward. Thus, *a.ppost_loc* is a conjunction where elementary constraints are either $(V'[i] = v)$, $(V'[i] = V[i] \quad op \quad v)$, or $(V'[i] = V[i])$. We denote by $Pre\_Const(V_{ctrl})$ (respectively $Post\_Const(V_{ctrl})$) the set of conditions respecting the syntax of *a.ppre_ctrl*

(respectively *a.ppost_ctrl*) on variables of the vector[1] V_{ctrl}.

To describe a use case, one has first to define mainly two parts: the system application domain and the set of the use case actions. The application domain consists of giving the state variable vectors and their respective possible values. At this stage, the analyst has to decide about the choice of the location variables and the control ones. Then, s/he expresses her/his use cases. We will describe the syntax of use cases in the next subsection.

We recall that our goal is to synthesize from the use cases an extended automaton which represents the behavior of the distributed system. Such automaton should be compatible with the description of use case actions given in definition 4:

Definition 5. *An extended automaton is a structure* $\mathcal{A} = (Loc, Loc_0, Var, Labels, T)$ *where:*

- Loc *is the set of locations of the extended automaton,*
- $Loc_0 \subset Loc$ *is the set of initial locations,*
- Var *is the set of variables,*
- $Labels$ *is the set of labels,*
- $T \subseteq Loc \times Pre\_Const(Var) \times Labels \times Post\_Const(Var) \times Loc$ *is the transition relation.*

An extended automaton behaves like a classical one, except that firing a transition is enabled by a precondition from $Pre\_Const(Var)$. Afterward, the system moves to the target location of the transition, and variables of the automaton are modified according to a post condition from $Post\_Const(Var)$.

[1] a vector is considered as set of variables

4.2.2 The Use Case Graph

The syntax of use case is shown in Figure 17. Use cases may enclose loops for which an optional maximum number of iterations may be specified.

In order to treat the textual use case description for deriving an automaton, we propose an intermediate form represented as use case graph (cf. Figure 18) constructed by parsing the use case textual description. The use case graph is composed of fictitious nodes connected by actions. *<Path>* in a node represents different alternatives while *<Loop>* is translated into a cycle in the use case graph. Finally, the initial node of the graph corresponds to the initial state of the use case. Formally, a use graph is defined as follows:

Definition 6. *A use case graph is a directed graph structure* $G = (N, N_0, E, A,$ *action*) *where:*

- $G.N$ *is the set of nodes,*
- $G.N_0 \subset G.N$ *is the set of initial nodes,*
- $G.E \subset G.N \times G.N$ *is the set of edges,*
- $G.A$ *is the set of the use case actions*

```
<use_case>      :: = begin_use_case <case_id>
                     (<Path> | <Loop>)+
                     end_use_case
<Path>          :: = begin_Path
                     <Action>+ (<Path> | <Loop>)*
                     end_Path
<Loop>          :: = begin_loop {INTEGER}
                     <Action_loop>+
                     end_loop
<Action_Loop>:: = begin_Act_Loop
                     <Path>* <Action>
                     end_Act_Loop
<Action>        :: = begin_Action
                     ppre_loc = <LOC_PRE_COND>
                     ppost_loc= <LOC_POST_COND>
                     lab= <LABEL>
                     ppre_ctrl =<CTRL_PRE_COND>
                     ppost_ctrl=<CTRL_POST_COND>
                     end_Action
```

Figure 17: Use Case Syntax

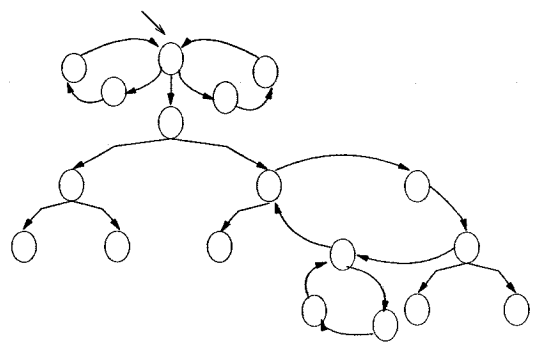


Figure 18: Use Case Graph

- *$G.action : G.E \mapsto G.A$ is a bijection that associates an action from $G.A$ to each edge in the graph.*

The use case syntax in Figure 17 says that the analyst-described use case has only a single initial node, but a use case may be transformed as we will see later into a graph which may have many initial nodes. Thus, definition 6 covers both cases. Since the syntax of use cases allows the explicit specification of loops, we define a loop structure which can be filled from the textual use case description.

Definition 7. *A loop $L = (id, iteration, init\_node, actions\_list)$ is a structure where: $L.id$ is the loop identifier, $L.init\_node$ is the starting node of the loop, $L.iteration$, if specified, is the maximum number of successive allowed iterations, and $L.actions\_list$ is a list of actions representing the body of the loop.*

4.3 Use Case Graph Transformation

It could be obvious to see that actions that are connected to a same node in a use case graph G should be sharing some conditions on location variables. However, this is not always the case because the user could have specified inconsistent information. In such situation, the system will never move forward from the inconsistent node. The problem is how to find a way that can help to point out such situation.

To this end, we define the context of a node as:

- **Initial Nodes:**

$$G.Context(n) = \bigvee_{(n,n') \in E} G.action(n, n').ppre\_loc$$

- **Intermidiate Nodes:**

$$G.Context(n) = \bigvee_{(n',n)\in E} G.action(n',n).ppost\_loc \\ \wedge \bigvee_{(n,n')\in E} G.action(n,n').ppre\_loc$$

- **Leaf Nodes:**

$$G.Context(n) = G.action(n',n).ppost\_loc$$

These formulas show that the context of the node preserves the conditions of its ingoing actions. The latter may be followed at least by one of its outgoing actions since the conjunction should not be empty. In the same way, the context of a node preserves the conditions of its outgoing actions that may be preceded at least by one of the ingoing actions to that node.

The context of a node can always be normalized in the form $C_1 \vee C_2 \vee \cdots \vee C_p$ where C_i matches a single location vector. However, our syntax does not allow disjunction in the location constraints. To that purpose, we introduce the notion of *Strongly Connected Node.* This notion concerns the initial node of the use case as well as its intermediate nodes but not leaf nodes because each has only one ingoing action and no outgoing actions.

Definition 8. *In a use case graph G, a node $n \in G.E$ is said to be a Strongly Connected Node (SCN) iff:* $\forall(n',n) \in G.E, \forall(n,n'') \in G.E$,

- **If n is an initial node then:**

$$G.Context(n) = action(n',n).ppre\_loc$$

- **If n is an intermediate node then:**

$$G.Context(n) = G.action(n', n).ppost\_loc = G.action(n, n'').ppre\_loc$$

A node n is `SCN` means that its context coincides exactly with both the partial location pre-conditions of its outgoing actions and, the partial location post-conditions of its ingoing actions. We observe that not all the nodes in the original user-described use case graph are `SCN`. Consequently, the graph G has to be modified by replacing all the non SCN nodes with a set of nodes that are `SCN`.

Figure 19 describes the algorithm of such transformation. The input of the transformation is a non SCN graph, and the output is a graph where all its nodes are SCN. Applying such transformation to the use case graph of Figure 20.(a) may result on the obtention of the same graph (Figure 20.(b)), the duplication of actions like in Figure 20.(c), or the elimination of those which can not be executed in such node (Figure 20.(d), and Figure 20.(e)). Duplication of actions will not cause any problems while the elimination may result in the introduction of some disconnectivities in the graph. Nevertheless, we believe that disconnectivies should be kept so that they have the possibility to be grafted in other use cases in the composition phase. Hence, their elimination may result in loss of information at this stage.

It is clear that an action can belong to at most one loop while a node can be shared by more than one since we do not accept the specification of embedded loops. The next step is to augment the current use case graph with the information of the remaining loops.

Originally, the loops are explicitly specified by the user, but by removing some actions, a specified cycle may be altered, and by duplication of actions loops can also be duplicated. Here, two solutions are to consider:

Input: a use case graph G and a non SCN node n
output: a use case graph G' where node n is replaced by SCN ones
Let G' be an empty use case graph
Let $Context(n) = C_1 \vee C_2 \vee \cdots \vee C_p$ where C_i matches a single location vector
For each C_i in $Context(n)$ do
  Let n_i be a new node
  For each n' such that $(n', n) \in G.E$ do
    $G'.N := G'.N \cup \{n_i\}$
    If $C_i \wedge action(n', n).ppost\_loc = C_i$ then
      $G'.N := G'.N \cup \{n'\}$
      $G'.E := G'.E \cup \{(n', n_i)\}$
      $G'.action(n', n_i).ppre\_loc := G.action(n, n_i).ppre\_loc$
      $G'.action(n', n_i).ppost\_loc := C_i$
      $G'.action(n', n_i).lab := G.action(n', n).lab$
      $G'.action(n', n_i).ppre\_ctrl := G.action(n', n).ppre\_ctrl$
      $G'.action(n', n_i).ppost\_ctrl := G.action(n', n).ppost\_ctrl$
      $G'.A := G'.A \cup \{G'.action(n', n_i)\}$
    done
  For each n' such $(n, n') \in G.E$ do
    $if\ C_i \wedge action(n, n').ppost\_loc := C_i\ then$
      $G'.N := G'.N \cup \{n'\}$
      $G'.E := G'.E \cup \{(n_i, n')\}$
      $G'.action(n_i, n').ppre\_loc := C_i$
      $G'.action(n_i, n').ppost\_loc := G.action(n, n').ppost\_loc$
      $G'.action(n_i, n').lab := G.action(n_i, n).lab$
      $G'.action(n_i, n').ppre\_ctrl := G.action(n', n).ppre\_ctrl =$
      $G'.action(n_i, n').ppost\_ctrl := G.action(n', n).ppost\_ctrl$
      $G'.A := G'.A \cup \{G'.action(n_i, n')\}$
    done
/* For the other nodes */ :
If $(n \neq n')$ and $(n \neq n")$ then
  $G'.N := G'.N \cup \{n' \in G.N | (n', n) \notin G'.E$ and $(n, n')\neg \in G'.E\}$
  $G'.E := G'.E \cup \{(n', n") \in G.N | (n \neq n')$ and $(n \neq n")\}$
  For all n' and $n"$ such that $(n \neq n')$ and $(n \neq n")$ and $(n', n") \in G.E$ do
    $G'.A := G'.A \cup \{G.action(n', n")\}$
    $G'.action(n', n") := G.action(n', n")$
  done
  if $n \in G.N_0$ then
    $G'.N_0 := \{n_1, n_2, \ldots, n_p\}$
    else $G'.N_0 := G.N_0$

Figure 19: `Transform_1` algorithm: Transforms a graph into a SCN graph

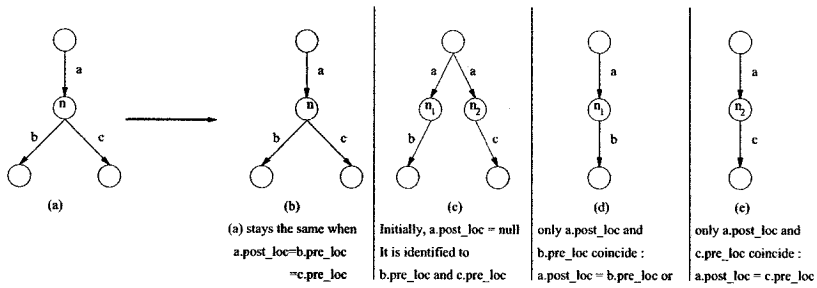


Figure 20: SCN Graph Derivation

1. either we suppose that all the user-specified loops are consistent, which means that none of their respective actions will be eliminated. Hence, we will find all the user-specified loops in the SCN graphs.

2. or we suppose that loops may be altered, an operation of retrieving the rest of these loops in the remaining ones has to be developed in consequence. It has as objective to make the bond between original loops and the ones in the SCN graph. When a loop in $G'.L_j$ is the same then the loop $G.L_i$, the $G'.L_j$ inherits the iteration number of $G.L_i$

Finally, as specified by the language, the loop information is not integrated in the definition of different actions and since loops are explicit, it is important that the automaton preserves the integrity of each loop. If we think in terms of behavior protection, leaving a loop in a non-expected manner, because of implied scenarios, may represent a failure in the system. From this view, we propose to protect all the originally specified loops. The algorithm for loop protection is described in Figure 21. Its input is a SCN graph where the loops are not decorated and the set of initially specified loops. It generates the same graph with protecting the loops by tagging the nodes that belongs to loops with the identifier of the loop.

Input : - $G' = (N, N_0, E, A, action)$ the use case graph where all nodes are SCN
- $L' = (id, init\_node, action\_list)$ a loop having a maximum number of iteration in the graph G'

Output : - modifies $G'.A$

Let $a = G'.action((L'.Init\_Node), n')$
such that $G'.action((L'.Init\_Node), n') \in L'.action\_list$
/* a is the first action of the loop*/
$a.ppre\_loc := a.ppre\_loc \wedge (Tag\_Lp = null)$
$a.ppost\_loc := a.ppost\_loc \wedge (Tag\_Lp = L'.id)$
$a.ppre\_ctrl := a.ppre\_ctrl \wedge (iter[L'.id] < L'.iteration)$
$a.ppost\_ctrl = a.ppost\_ctrl \wedge (iter'[L'.id] := iter[L'] + 1)$
Let $tmp\_action\_list = L'.action\_list$
$tmp\_action\_list := tmp\_action\_list - \{a\}$
$Update(n', L'.action\_list, (Tag\_Lp = L'.id))$

For each n' such that $(n', L'.Init\_Node) \in G'.E$
and $G'.action(n', L'.Init\_Node) \notin L'.action\_list$ do
Let $a = G'.action(n', L'.Init\_Node)$
$a.ppre\_loc := a.ppre\_loc \wedge (Tag\_Lp = null)$
$a.ppost\_loc := a.ppost\_loc \wedge (Tag\_Lp = null)$
$a.ppost\_ctrl := a.ppost\_ctrl \wedge (iter'[L'.id] = 0)$
done

Let $a \in L'.action\_list$ such that $a = G'.action(n', L'.Init\_Node) \in G'.E$
/* a is the last action of the loop*/
$a.ppre\_loc := a.ppre\_loc \wedge (Tag\_Lp = L'.id)$
$a.ppost\_loc := a.ppost\_loc \wedge (Tag\_Lp = null)$
$tmp\_action\_list := tmp\_action\_list - \{a\}$

For each $(n, n') \in G'.E$ and $G'.action(n, n') \in tmp\_action\_list$ do
/* the remainder of the action list */
Let $a = G'.action(n, n')$
$a.ppre\_loc := a.ppre\_loc \wedge (Tag\_Lp = L'.id)$
$a.ppost\_loc := a.ppost\_loc \wedge (Tag\_Lp = L'.id)$
$Update(n', L'.action\_list, (Tag\_Lp = v)$
done

For each $(n, n') \in G'.E$ do
Let $a = G'.action(n, n')$
/*$(Tag\_Lp = null)$ is neutral if Tag_Lp has already been set to another value*/
$a.ppre\_loc := a.ppre\_loc \wedge (Tag\_Lp = null)$
$a.ppost\_loc := a.ppost\_loc \wedge (Tag\_Lp = null)$
done

Where $Update(n, Act\_List, cond)$ means
For each $(n, n') \in G'.E$ such that $G'.action(n, n') \notin Act\_List$ do
Let $a = G'.action(n, n')$
$a.ppre\_loc := a.ppre\_loc \wedge cond$
$a.ppost\_loc := a.ppost\_loc \wedge (Tag\_Lp = null)$
done

Figure 21: `Transform_2` algorithm: Loop protection of SCN graph

4.4 Use Cases Integration

Let's now assemble the pieces of the puzzle. Our goal is to synthesize an automaton from given use cases. As shown in Figure 22, the synthesis of the automaton begins by treating the nodes on the original graph use case so that they obey to the syntax of the language by making them SCN. Next, the information on the loops has to be integrated in the action specification. Now, the graphs resulting from each use case at this stage are merged together. Let G_1 and G_2 be two use cases graphs having the structure $(N, N_0, E, A, \ action)$. The union graph of G_1 and G_2 is defined by:

- $(G_1 \cup G_2).N = G_1.N \cup G_2.N$
- $(G_1 \cup G_2).N_0 = G_1.N_0 \cup G_2.N_0$
- $(G_1 \cup G_2).A = G_1.A \cup G_2.A$
- $(G_1 \cup G_2).E = G_1.E \cup G_2.E$
- $(G_1 \cup G_2).action$ is defined by:

$$(G_1 \cup G_2).action(n, n') = \begin{cases} G_1.action(n, n') & if (n, n') \in G_1.E \\ G_2.action(n, n') & if (n, n') \in G_2.E \end{cases}$$

At this stage the synthesis of an automaton from use cases becomes straightforward. In order to get the overall behavior of the system, one has to integrate many use cases. Let G be the union graph of the given use cases. $\mathcal{A} = (Loc, Loc_0, Var, \ Labels, T)$ the automaton of those use cases derived from G is defined by:

- $Loc = \{G.context(n) | n \in G.N\}$
- $Loc_0 = \{G.context(n) | n \in G.N_0\}$

- $Var = V_{ctrl}$
- $Labels = \{a.lab | a \in G.A\}$
- $T = \{(a.ppre\_loc, a.ppre\_ctrl, a.lab, a.ppost\_ctrl, a.ppost\_loc) | a \in G.A\}$

We notice that all the node in G are assumed to be SCN. This is due to the fact that the composed graphs are originally SCN. Consequently, we have ($G.context(n) = a.ppre\_loc = a.ppost\_loc$).

We consider that the initial locations of the automaton are the initial nodes of all the use cases because the analyst would not commit at the first stage to a specific choice. However, she/he may decide later which ones to keep. As our goal is to synthesis from use cases an automaton, the latter may contain inconsistencies that reflect the behavior of the given use cases. For instance, the automaton may be disconnected. Such anomaly may happen due to either a contradiction or an omission in the specified use cases. In addition, since loops are explicitly specified, we have protected them so that no implied scenarios will interfere with their behavior. We notice that we can apply the same approach to protect interference between use cases.

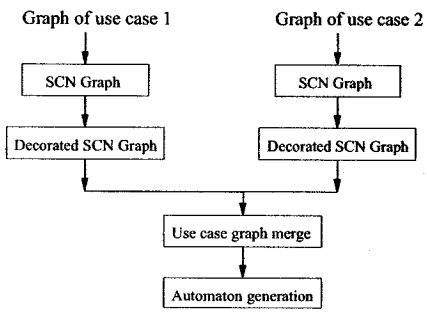


Figure 22: The Automaton Generation Process

4.5 Application of the Composition Approach to the Alternating Bit Protocol

The Alternating Bit Protocol (ABP) is used to guarantee correct data reception between a sender A and a receiver B using a channel that loses or corrupts messages. The approach consists of extending messages with one bit, which will be alternated only when messages are correctly received by the other side.

Hence, A assumes that the data is successfully received by the other side provided that it receives an acknowledgment with the expected bit value. A sends the same message again if the received acknowledgment is corrupted or after a timer expiration. In contrast, B acknowledges the reception of a message if the checksum of the data is valid. We also assume that a message cannot be sent more than three times. After what, the channel is supposed to be down.

The ABP V_{loc} is composed by the Sender State Variable St_A, the Receiver state variable St_B and the bit value when sending the message Bit. The control vector is composed by the timing out variable To, the acknowledgment variable Ack, and the checksum variable $CheckSum$. We describe the behavior of the previous protocol when the bit is set to 0 by two use cases. The first one is shown in Figure 23.a and treats the reception of corrupted and non-corrupted data in side B. Figure 23.b represents the use case of the acknowledgment reception. The actions set of the two use cases are respectively given in Tables 2 and 3.

We can see in use case Table 2 that the checksum variable was not initiated because it could not be known before sending the information. There is an action in the receiver side that verifies this checking and according to it the system will move to one of the two directions. To apply the approach described above, variables $iter$ and Tag_Lp are added

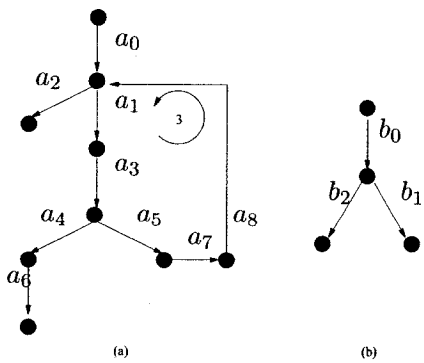


Figure 23: Original Described Use Case Graphs

to the state variable vector to take into account loops. The actions in Table 4 represent the ones from which the use case automaton is derived. The latter is shown at Figure 24. Without the tagging we are doing, the two dashed automaton locations would be merged. This introduces an implied scenario that shows that a verification of a checksum can be repeated indefinitely causing a multiple acknowledgment sending for the same data, annoying the loop behavior specified by the user.

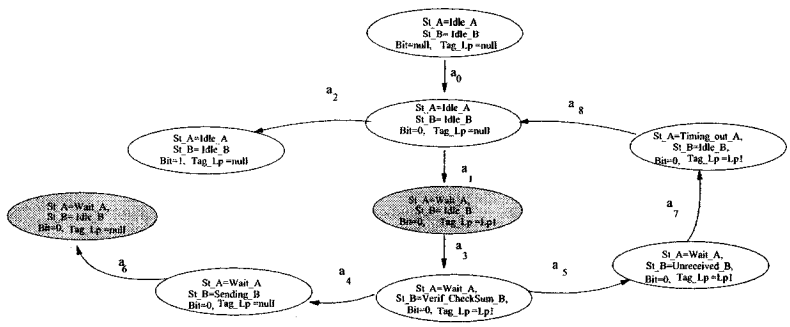


Figure 24: Automaton of the Use Case Corrupted and Non-corrupted Data Reception

The overall system automaton is shown in 25.

Table 2: Use Case Actions of Corrupted and Non-Corrupted Data Reception

Actions	*ppre_loc*	*ppre_contr*	*lab*	*ppost_contr*	*ppost_loc*
a_0	St_A =Idle_A ∧ St_B =Idle_B ∧ Bit= null	To = false ∧ CheckSum=null	Initiate_Data_0_send	To= false ∧ Checksum = null	St_A =Wait_A ∧ St_B =Idle_B ∧ Bit= 0
a_1	St_A = Idle_A ∧ St_B =Idle_B ∧ Bit= 0	To= false ∧ CheckSum =null	Send_Data	To= false ∧ CheckSum=null	St_A = Wait_A ∧ St_B =Idle_B ∧ Bit= 0
a_2	St_A =Idle_A ∧ ∧ Bit= 0 St_B =Idle_B	To=false ∧ CheckSum=null	Alternate_Bit	To= false ∧ CheckSum=null CheckSum=null	St_A = Idle_A ∧ ∧ Bit= 1 St_B =Idle_B
a_3	St_A = Wait_A ∧ ∧ Bit= 0 St_B = Idle_B	To= false ∧ CheckSum=null	Receive_Data	To= false ∧ CheckSum=null	St_A = Wait_A ∧ ∧ Bit= 0 St_B =Verifying_B
a_4	St_A = Wait_A ∧ ∧ Bit= 0 St_B = Verifying_B	To= false ∧ CheckSum=null	Verify_Checksum	To= false ∧ CheckSum= True	St_A = Wait_A ∧ ∧ Bit= 0 St_B =Sending_B
a_5	St_A =Wait_A ∧ ∧ Bit= 0 St_B =Verifying_B	To= false ∧ CheckSum=null	Verify_CheckSum	To= false ∧ CheckSum= false	St_A = Wait_A ∧ ∧ Bit= 0 St_B =Unreceived_B
a_6	St_A = Wait_A ∧ ∧ Bit= 0 St_B =Sending_B	To= false ∧ CheckSum=True	Send_Ack	To= false ∧ CheckSum= null	St_A = Wait_A ∧ ∧ Bit= 0 St_B =Idle_B
a_7	St_A = Wait_A ∧ ∧ Bit= 0 St_B =Unreceived_B	To= false ∧ CheckSum= false	Time_Out	To= true ∧ CheckSum= null	St_A = Timing_out_A ∧ ∧ Bit= 0 St_B =Idle_B
a_8	St_A = Timing_out_A ∧ ∧ Bit= 0 St_B =Idle_B	To= true ∧ CheckSum=false	Initiate_Send	To= false ∧ CheckSum= null	St_A = Idle_A ∧ ∧ Bit= 0 St_B = Idle_B

4.6 Summary

In this chapter, we presented a model for use case description based on actions and an approach for implicit composition of use cases. It consists of enriching the model in [91] with control variables and explicit loops, fact that gives the language more expressiveness.

From the textual description of the use case, we generate its representative graph. The latter has to be modified so that the location condition of each action enables its execution, eliminating hence the (semantically) incoherent ones. Second, we ensure the implementation of the loop information in the concerned actions of the graph. This implies systematically the protection of loops, which preserves their integrity when merging different use cases. At the end, the automaton of the use case composition is derived.

This approach is very interesting when it comes to the specification of overlapping use

Table 3: Actions of the Acknowledgment Reception Use Case

Rules	*pre_loc*	*pre_contr*	*lab*	*post_contr*	*post_loc*
b_0	(St_A= Wait_A ∧ (∧ Bit= 0) St_B= Idle_B)	(To= false ∧ (∧ ack=Null) CheckSum= True)	(Receive_Ack)	(To= false ∧ (∧ ack=Null) (CheckSum= Null)	(St_A= Verifying_A ∧ (∧ Bit= 0) St_B= Idle_B)
b_1	(St_A= Verifying_A ∧ (∧ Bit= 0) St_B= Idle_B)	(To= false ∧ (∧ ack=Null) CheckSum= Null)	(Verify_Ack_)	(To= false ∧ (∧ Ack=True) CheckSum=Null)	(St_A=Idle_A ∧ (∧ Bit= 1) St_B= Idle_B)
b_2	(St_A=Verifying_A ∧ (∧ Bit= 0) St_B= Idle_B)	(To= false ∧ (∧ ack=Null) CheckSum= Null)	(Verify_Ack_)	(To= false ∧ (∧ Ack=false) CheckSum=Null)	(St_A= Idle_A ∧ (∧ Bit= 0) St_B= Idle_B)

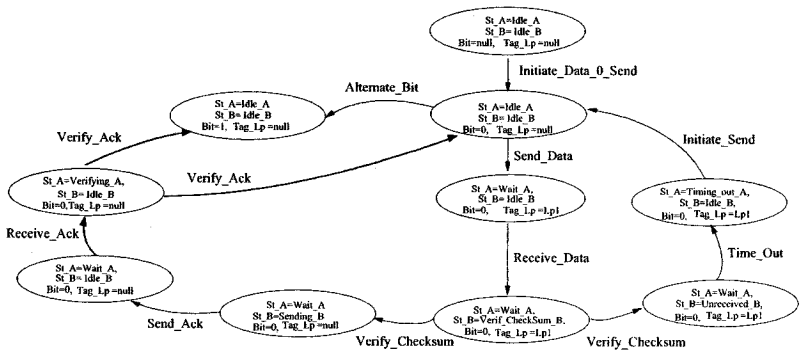


Figure 25: Automaton of Use Case Integration

cases. Designers can provide requirements which are not necessary independent. When composing implicitly use cases, the overlapped parts will be merged. It is a way to generate an overall system behavior but also to complete the views of the different stakeholders.

Despite the expressiveness the approach gave, the composition of use cases presents some limitations, mainly:

- **The non assurance of the connectivity of the resulting automaton** :

  Because of the transformations we are processing before composing, the connectivity of the resulting automaton is no more guaranteed. In fact, some connected actions may become unconnected because of the context of the reached state. It is a matter of having a certain consistency with what the modeler has described. We assumed that a disconnected parts can be connected in another stage of composition since the

Table 4: Use Case Actions of Corrupted and Non-Corrupted Data Reception after `Transform_1` and `Transform_2`

Actions	*ppre_loc*	*ppre_contr*	*lab*	*ppost_contr*	*ppost_loc*
a_0	St_A =Idle_A ∧ St_B =Idle_B ∧ Bit= null ∧ *Tag_Lp*= null	To = false ∧ CheckSum=null	Initiate_Data_0_send	To= false ∧ Checksum = null ∧ iter'[Lp_1]=0	St_A =Wait_A ∧ St_B =Idle_B ∧ Bit= 0 ∧ *Tag_Lp*= null
a_1	St_A = Idle_A ∧ St_B =Idle_B ∧ Bit= 0 ∧ *Tag_Lp*= null	To= false∧ CheckSum =null ∧ iter[Lp_1] < 3	Send_Data	To= false ∧ CheckSum=null ∧ iter'[Lp_1]=iter[Lp_1]+1	St_A = Wait_A ∧ St_B =Idle_B ∧ Bit= 0 *Tag_Lp*= Lp_1
a_2	St_A =Idle_A ∧ St_B =Idle_B ∧ Bit= 0 ∧ *Tag_Lp*= null	CheckSum=null ∧ CheckSum=null	Alternate_Bit	To= false ∧ CheckSum=null	St_A = Idle_A ∧ St_B =Idle_B ∧ Bit= 1 ∧ *Tag_Lp*= null
a_3	St_A = Wait_A ∧ St_B = Idle_B ∧ Bit= 0 ∧ *Tag_Lp* = Lp_1	To= false ∧ CheckSum=null	Receive_Data	To= false ∧ CheckSum=null	St_A = Wait_A ∧ St_B =Verifying_B ∧ Bit= 0 ∧ *Tag_Lp*= Lp_1
a_4	St_A = Wait_A ∧ St_B = Verifying_B ∧ Bit= 0 ∧ *Tag_Lp* = Lp_1	To= false ∧ CheckSum=null	Verify_Checksum	To= false ∧ CheckSum= True	St_A = Wait_A ∧ St_B =Sending_B ∧ Bit= 0 ∧ *Tag_Lp*= null
a_5	St_A =Wait_A ∧ St_B =Verifying_B ∧ Bit= 0 ∧ *Tag_Lp*= Lp_1	To= false ∧ CheckSum=null	Verify_CheckSum	To= false ∧ CheckSum= false	St_A = Wait_A ∧ St_B =Unreceived_B ∧ Bit= 0∧ *Tag_Lp*= Lp_1
a_6	St_A = Wait_A ∧ St_B =Sending_B ∧ Bit= 0 ∧ *Tag_Lp*= null	To= false ∧ CheckSum=True	Send_Ack	To= false ∧ CheckSum= null	St_A = Wait_A ∧ St_B =Idle_B ∧ Bit= 0 ∧ *Tag_Lp*= null
a_7	St_A = Wait_A ∧ St_B =Unreceived_B ∧ Bit= 0 ∧ *Tag_Lp*= Lp_1	To= false ∧ CheckSum= false	Time_Out	To= true ∧ CheckSum= null	St_A = Timing_out_A ∧ St_B =Idle_B ∧ Bit= 0 ∧ *Tag_Lp*= Lp_1
a_8	St_A = Timing_out_A ∧ St_B =Idle_B ∧ Bit= 0 ∧ *Tag_Lp*= Lp_1	To= true ∧ CheckSum=false	Initiate_Send	To= false ∧ CheckSum= null	St_A = Idle_A ∧ St_B = Idle_B ∧ Bit= 0 ∧ *Tag_Lp*= null

integration is based on merging the states having the same characterization. However, at the end, some parts may still be disconnected, which may denote an inconsistencies in the whole specification. This problem can be solved when we consider the case where the analyst describes consistent use cases. Consistent use case denotes here the fact that all the ingoing actions to a state are executable on that state.

- **Implied scenarios:**

  As mentioned earlier, implicit composition may lead to the appearance of implied scenarios. We are not escaping to this rule with the approach we are proposing. By protecting the loops when composing, we are reducing the number of implied scenarios.

However, we are not eliminating all of them. As a solution, a tagging of the actions with the identifier of the use case they belong to can be made.

While the proposed approach is suitable for overlapping use cases, it is not the case when the designer needs to explicitly specify the relationships between use cases. In the next chapter, we will enrich the same model of use cases in order to handle explicit composition. The overlapping behavior will no more be considered as a criterion of composition.

Chapter 5

Explicit Composition of Use Cases using Interactions

5.1 Introduction

Nowadays, separation of concerns is at the core of software and system engineering. It refers to the ability to identify, encapsulate, and manipulate parts of software that are relevant to a particular functionality, task, or purpose. As specified in [54], separation of concerns is one of the most common techniques in the software/system engineering. Concerns are the primary motivation for organizing and decomposing software/system description into smaller and comprehensible parts, each of which addresses one or more concerns.

Use cases is one of the techniques used to express concerns [54]. In the opposite to what was presented in the previous chapter, when modeling, the analyst will not think in terms of system traces to merge, but in terms of (1) What concerns should I model, (2) How can I separate them, and (3) How should I compose them.

[1]Published in [69] and in [76]

In this chapter we present a composition approach for merging use cases based on the *interactions* they are making between each other. Our main objective is to make the composition easier by synthesizing automatically an automaton of the system without introducing any implied scenarios starting from a set of use cases that represent different concerns. Use cases are not manipulated as blocks. A new level of granularity is defined, where each use case can be inserted within another use case according to some semantics.

Our model of use case is an extended automaton enriched with interactions, where an interaction is an invocation of a use case by another. Using state-based patterns, interactions specified within the use case are translated into a state-based model, connecting the respective automata of the interacting use cases. In this context, a pattern defines the state-based semantics of a use case interaction. To avoid implied scenarios, we propose to build a *use case interaction graph* from system use cases. It is used to detect the potential implied use case interactions, which we call *interferences.* Additional constraints, expressed in terms of variables, are added to the system automaton to eliminate such interferences.

The chapter is structured as follows.

- Section 5.2 gives an overview of the use case model we are using.
- Section 5.3 describes the synthesis of the overall automaton of the system based on the interactions between use cases.
- Section 5.4 presents an application of our approach on use cases of an e-Purchasing system.
- Section 5.5 summarizes the chapter.
- Finally, Section 5.6 gives the motivation of having such composition approach.

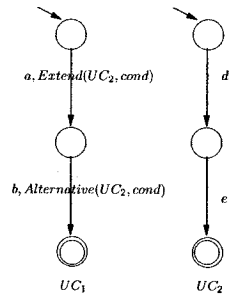


Figure 26: Use Case model with interactions

5.2 Use Case Acquisition : Model Presentation

In order to describe the functional requirements of a system, the designer needs to specify a set of use cases and interactions which exist between use cases. Offering a notation that gives the possibility to express these interactions simplifies the specification of use cases and promotes reusability, which is one of the main goals of scenario-based approaches. In fact, when a specific behavior is needed and referenced by different use cases, a separate use case can be specified to encapsulate this behavior. This use case will be invoked within the other use cases each time the behavior is required, which helps avoiding redundancy in the specification.

In order to specify interactions within the use case description, we extend the model given in chapter 4 section 4.2 with interactions. The analyst will have the possibility to associate an interaction with the label of an action, as shown in Figure 26. Hence, the model of use cases will be an extended finite state automaton with interactions.

In this chapter, we present the two most needed interactions between use cases, namely `Extend`, and `Alternative`.

- `Extend(`*uc*`,`*cond*`)` specifies an optional invocation of a use case *uc* after firing the transition where the interaction is specified. Hence, when a use case uc_1 specifies

an invocation of the use case *uc* with the `Extend` interaction, *uc* will be executed only when *cond* is satisfied. *uc* is effectively an alternative course of the use case uc_1 after the transition to which it is associated. After finishing the execution of this alternative, the system resumes back and executes the rest of uc_1.

Furthermore, `Extend(`*uc*`,`*true*`)` specifies a mandatory invocation of a use case *uc* since the condition *cond* will always be satisfied, and hence the base use case will always invoke the use case *uc*. We denote it by `Include(`*uc*`)` in order to distinguish this case. Finally, we note that if `Include(`*uc*`)` invocation is done at the end of a use case, then it simply expresses the sequential concatenation.

- `Alternative(`*uc*`,` *cond*`)` specifies an interrupting use case *uc*. It is used to state explicitly the actions where an interruption of the current use case can be performed. Unlike the previous interactions, `Alternative` does not resume back the execution of the base use case after the execution of the interrupting one.

An interaction between use cases is expressed in the base use case in the form $(a, interaction)$ where $interaction \in \{$ `Include(`*uc*`)`, `Alternative(`*uc*`,`*cond*`)`, `Extend(`*uc*`,`*cond*`)` $\}$, and a represents the action preceding the interaction. A unique interaction can be specified with a given action.

5.3 Integration Approach

We define a two-step automated approach to produce an automaton modeling the overall behavior of the system from a set of use cases, avoiding unintended behaviors. The first step consists of translating the use case interactions into a state-based model. It is achieved by means of state-based patterns for each interaction type. Hence, the structure of the

obtained automaton reflects the behaviors from specified interactions. The second step consists of deriving a use case interaction graph, used to detect cycles and interferences between use cases and their interactions.

5.3.1 State-Based Synthesis Patterns of Use Case Interactions

Interaction patterns serve the automated merger of the use case automata into an overall system automaton. They provide the join points between the use case automata during their integration. Figure 27 shows the integration patterns of `Include`, `Extend`, and `Alternative` invocations. For automaton minimization, and since a transition with *false* condition will never be fired, we represent the pattern of `Include` (Figure 27). Transformations are applied to the automaton of the base use case: transitions are added to refer to the initial and final locations in the use case specified in the interaction. *start_uc* and *return_uc* are two new special labels used to connect use cases together. The condition given within an `Extend` or an `Alternative` interaction is a precondition of the added transition *start_uc*. Additional pre conditions may be added to the actions labeled with *start_uc* and *return_uc* according to the use case interaction graph as we will show later on. An ϵ-transition is also added and controlled with the negation of the condition the interaction specifies. It is fired when the condition of the invocation is not verified and the execution should not go through the called use case. The derived automaton after interpretation of the use case interaction is called *Interaction-Free* use case automaton since its edges are no more labeled by interactions.

5.3.2 Use Case Interaction Graph

During the specification of the use cases, the designer has defined different interactions within the use cases. However, since it is difficult to have the big picture at this stage, it

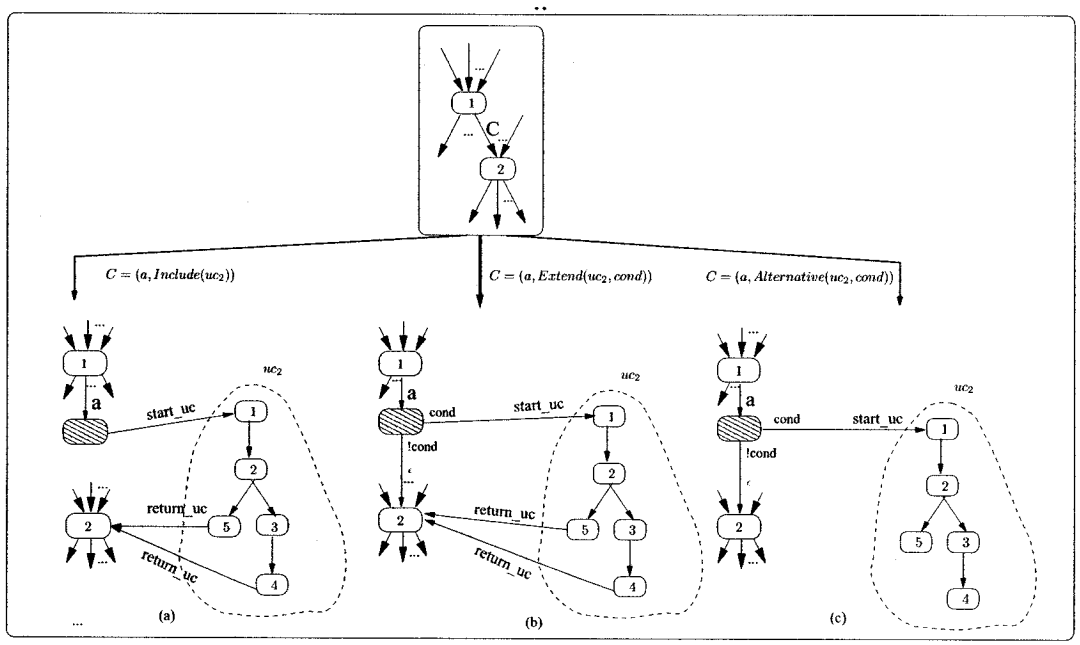


Figure 27: State-Based Synthesis Patterns of use case Interactions

could happen that the specified use case interactions generate interferences which may lead to unexpected scenarios in the overall system behavior.

Let's consider the case of two use cases making an `Include` invocation to the same use case (c.f. Figure 28 (a) and (b)). After the application of the state-based synthesis pattern, the system may eventually run through (a, uc_1, d). The same anomaly appears when a use case makes multiple invocations to the same use case (c.f. Figure 28) in different states.

In order to detect such behaviors, we propose to synthesize and analyze a *use case interaction graph* which is a directed graph with nodes representing use cases and edges linking together two interacting nodes of use cases. The edges are labeled with the type of the interactions that the base use case is performing as well as the number of occurrence of each invocation. The label $(Include, 2)$ in Figure (28 (c)) indicates that the use case uc_1 interacts twice with the use case uc_2. Unlike the UML use case diagram, our use case

interaction graph is build from the set of use case automata specified by the designer and models their interactions within their structure.

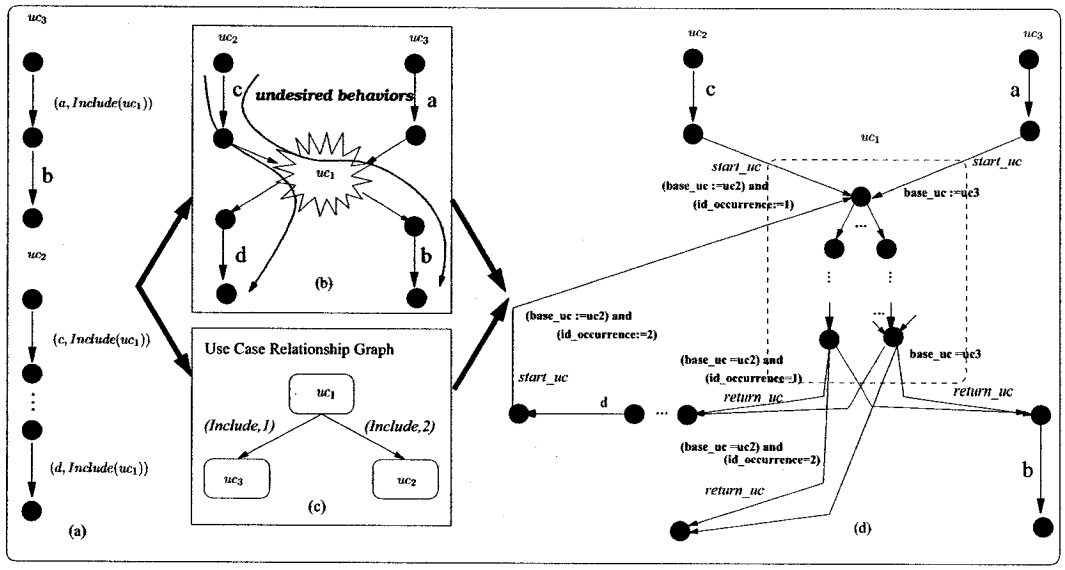


Figure 28: Inter and Intra-Implied Scenarios Resulting from Use case Merging

Forbidden Cycle Detection

The first step is based on using the use case interaction graph to detect potential problematic interactions in the original use case specification. Situations like a self call in a use case or a mutual calls between two use cases may lead to disconnected or non-executable part of the automaton in the system specification. Such situation indicates an eventual omission of some use cases or/and interactions or errors, which should be prevented. It is clear that a cycle where the edges are labelled only with the *Include* interaction are not intended in any use case specification. It results automatically to a non-executable part in the system automaton. In our approach, we propose to verify the existence of such cycle and revising the original specification consequently. In addition, the designer may specify some other forbidden cycles from the beginning. We propose to have an acyclic use case interaction

graph before the generation of the use case automaton which guarantees the prevention of disconnections in the final system automaton.

Interference Detection

The second step is based on detecting and eliminating implied scenarios arose from interferences between interactions. For this purpose, we propose to over-constrain the system automaton when composing the use cases with conditions on variables. We distinguish two cases where variables have to be added to the automaton of the system:

- when a use case is referred by many distinct use cases, independently from the type of the interaction, inter-implied scenarios may occur. A new variable such as `base_uc`, is added to the automaton. `base_uc` is set by transition *start_uc* to the identifier of the base use case (c.f. Figure (28 (d)). The *return_uc* transition will check the value of the `base_uc` variable.

- when a use case is referred more than once within the same use case, intra-implied scenarios may occur. A new variable such as `id_occurrence`, is added to the automaton. `id_occurrence` is set with the identifier of the occurrence in the base use case by the transition *start_uc*. In the same way, the *return_uc* transition will check the value of the `id_occurrence` variable (c.f. Figure (28 (d)).

Interaction `Alternative` is an exception to the previous rules: if a use case is only referred through `Alternative`, no implied behavior would be added as there is no return to the base use case. Therefore, the number of variables added to prevent implied scenarios depend on the existing interactions between the original use cases.

After the application of the state-based synthesis patterns and the addition of constraints to avoid implied scenarios, the composition of the interaction-free use case automata would represent the final automaton of the system (c.f. Figure 29). The initial states of the automaton of the system need to be specified by the user.

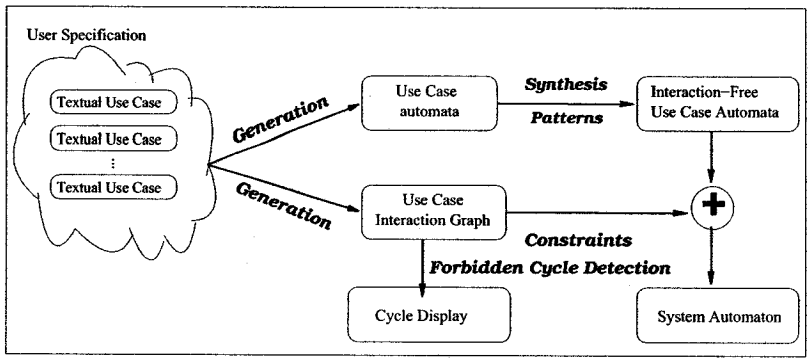


Figure 29: Overview of the Interaction-Based Integration Approach

5.4 Application on an e-Purchasing System

We apply our interaction-based integration approach to synthesize the specification of an e-Purchasing system. The functional requirements of an e-Purchasing system include many use cases. We focus particularly on these use cases: *"Register_Order"*, *"Login"*, *"Verify_Inventory"*, *"Print_Out"*, and *"Cancel_Order"*.

The use case *"Register_Order"* invokes use case *"Login"* in order to perform the authentication of a user when the user wants to make an order. Use case *"Register_Order"* also makes an optional interaction with use case *"Print_Out"* whenever the user requests to print out his order or the inventory. *Ccl*, *pr*, *inventory* and *attempt* are the variables of the system. The two first ones are used for the conditional execution of the two use case *"Cancel_Order"* and *"Print_Out"* respectively. The third one shows the availability of the product while the

fourth one counts the number of attempts to login. We present in Figure 30 the extended use case automaton of *"Register_Order"*. The label *RO* indicates that the location belongs to the automaton of use case *"Register_Order"*. The same notations are used for the other use cases. Hatched locations represent locations added to insert *start_uc* and *return_uc* transitions.

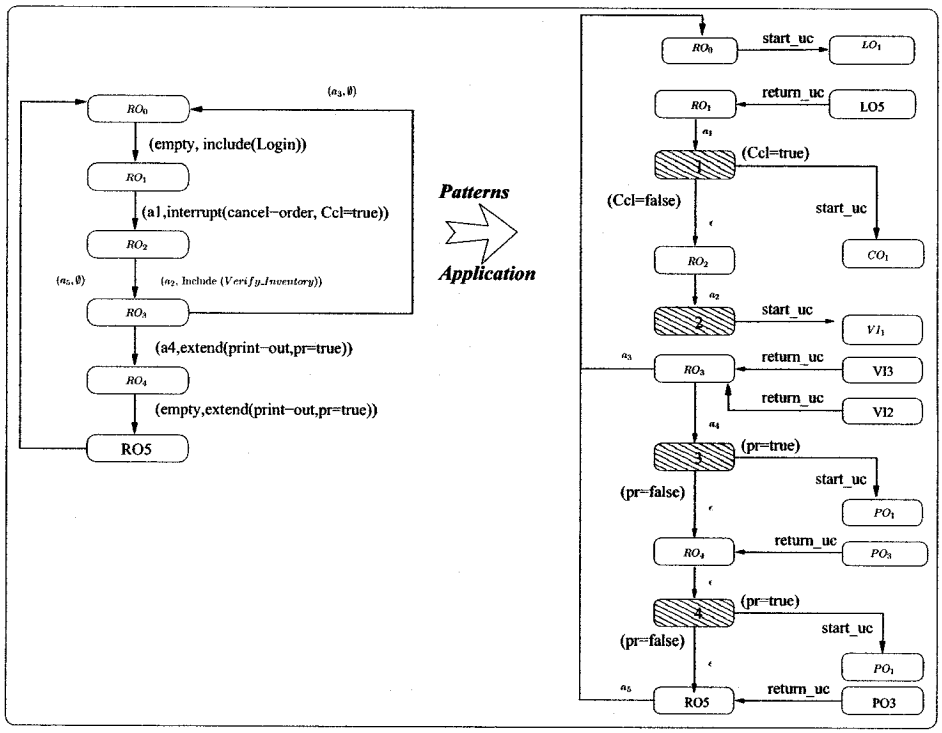


Figure 30: The Interaction-Free Automaton of the *Register_Order* Use Case

Figure 31 shows the use case interaction graph of the e-Purchasing system. We note that there is no `Include` cycle. Hence, no revision of the original specification is needed. However, interferences do exist. *"Print_Order"* is called twice in the use case *"Register_Order"*. Furthermore, it is called by two different use cases *"Register_Order"* and *"Cancel_Order"*.

According to the use case interaction graph, we need to add constraints on two variables, *entering_po* and *id_call*. As *"Print_Out"* can be called by *"Register_Order"* and *"Cancel_Order"*, variable *entering_po* serves to keep track of the base use case. Variable *id_call* keeps the location from which *"Register_Order"* called *"Print_Out"*. The automaton of the whole specification is given in Figure 32. Locations with identical labels are the same, but have been duplicated for clarity.

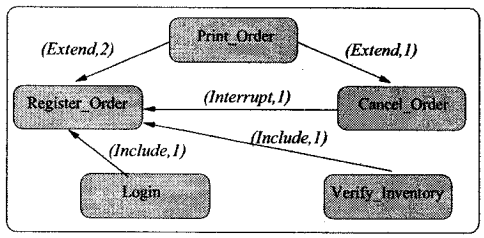


Figure 31: The Use case Interaction Graph of the e-Purchasing Use Cases

5.5 Summary

In this chapter, we have presented an approach that automatically generates a state-based model of the system from a set of use cases and their interactions. Instead of traditional composition methods based on the well-known constructors such as sequencing, alternative, and iterations, we proposed a composition approach based on inter-use case interactions. For this purpose, we have developed a two-step composition approach. The first step consists of generating an interaction-free use case automata using the state-based synthesis patterns of the different interactions. The second step analyzes the interactions between use cases in order to detect inconsistencies and implied behaviors in the specification. We proposed to synthesize a use case interaction graph for this purpose. Since the existence of

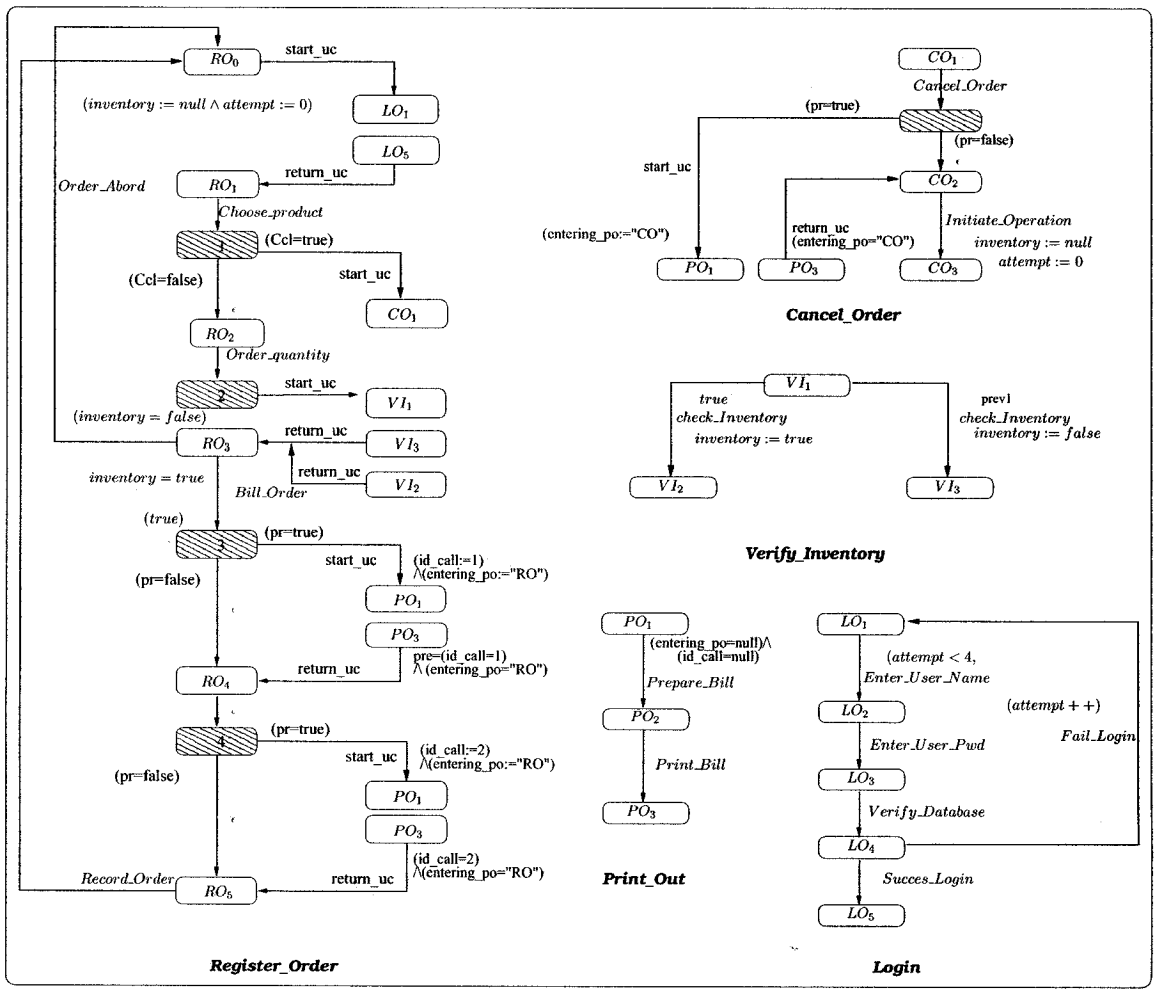


Figure 32: The e-Purchasing System Automaton

cycles within the graph expresses potential inconsistencies in the specification, we prevented cycles of the *include* interaction. To avoid interferences in the specification, we have added variables to the system automaton. The number of added variables is determined according to the interaction graph.

5.6 Strengths and Weaknesses of the Proposed Use Case Model and Composition Approach

On the opposite to what we have proposed in the first chapter, we are concentrating on the composition of independent use cases. The interactions specified within the use cases themselves are considered as the composition criterion.

This approach has many advantages:

- First, it promotes a modeling that separates the different concerns of the system, one of the commonly used technique in requirements analysis.
- Second, the interactions are specified within the use cases themselves while the traditional approaches use conventional operators -such as sequential, alternative, and iterations- that specify how use cases are linked together.
- Third, interferences between use cases are detected. We use variables in order to avoid implied scenarios (due to interference) generated by composition. In order to add new functionalities, the designer will not worry about the overall picture of the system structure, focusing on partial behaviors while our approach will take care of the verification of their interactions with the existing ones.

Despite the interesting results we obtained, our approach still needs some improvements:

- Update of use cases:

  As described, an interaction is related to an action. Hence, once the specification of use cases is achieved, there is no way to add new interactions between use cases

unless by re-describing the existing set use cases. Hence, our approach does not yet facilitate the maintainability of the overall specification, one of the main objectives of this thesis. Although the approach is fully automated, it is not incremental, as it is supposed to be in the requirements engineering phase. A change in the needs of the stakeholders requires the redefinition of some use cases and another run of the composition approach.

- Control variable number :

  By removing the implied scenarios, we are increasing considerably the number of variables of the system automaton, especially with a large number of use cases.

- Dependency between use cases:

  By composing two use cases, a link is set between these two use cases. The addition of transition labeled by *start_uc* and *return_uc* links forever the two use cases. Consequently, the original use cases do no more exist. Only the generated use case is kept because of the cut and paste operation we are performing. Such approach is not always desirable. Keeping the original use cases for further composition may be useful.

In the next chapter, we present a novel composition approach that overcome these shortcomings. It represents a formalization of the presented approach along with an improvement of the composition limitations discussed previously.

Chapter 6

Explicit Composition of Use Cases: A Novel Methodology using Imperative Expressions

6.1 Introduction

In the literature, identification of common states in different partial behaviors is commonly used as a criterion of composition [39, 101]. However, this criterion has the drawback of assuming that the designer has a deep understanding of the system's behavior. S/he should have an overall view of the intended behavior so that she/he is able to accurately define the naming of states in different use cases. This is a very demanding task if we consider the incremental nature of the definition of the system behavior itself. In addition, use cases are supposed to be defined independently and not necessary by the same designer. The

[1]Published in [75]

set of use cases may represent a collection of use cases described by different modelers. Composing use cases using explicitly defined relationships is more adapted to be used in an incremental process of generating a system specification from a set of use cases.

As stated in the previous chapter, specifying the relationship between a use case and another within the description of the use case itself imposes some limitations. Adding new relationships needs the redefinition of the use case. Defining independent composition operators that have specific composition semantics would facilitate the incremental elaboration of the overall system behavior.

In this chapter, we propose a new methodology for specifying and composing use cases. It is based on the definition of a set of use cases and composition expressions that shows the way use cases has to be composed.

The chapter is structured as follows:

- Section 6.2 gives an overview of the approach in the frame of requirements elicitation process.
- Section 6.3 presents the composition expression syntax as well as the different operators we are defining.

6.2 Approach Description

To formalize the informal aspects of use cases, we developed an automated and incremental approach for elaborating a state based model of the intended system behavior. From informal requirements, the modeler has to generate formal use cases. These use cases are composed in a formal and incremental way in order to generate the system specification expressed as a state-based model. This model can be checked against consistency using

model checking and/or simulation, as shown in Figure 33.

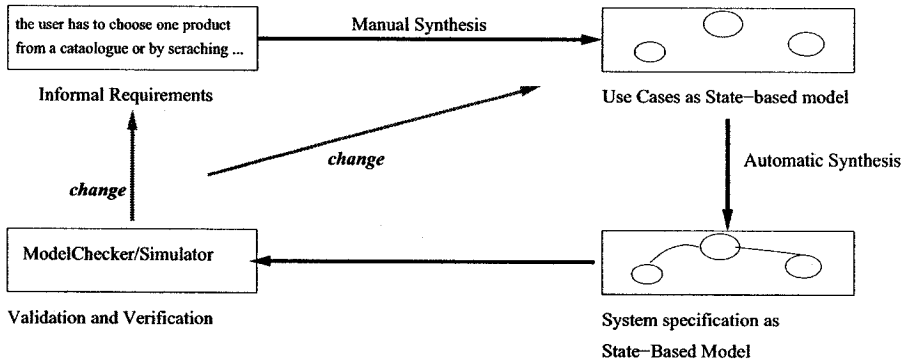


Figure 33: Approach Overview in Requirements Engineering

Our approach of composition consists of three main steps. First, the analyst provides a set of use cases where each one defines a partial system behavior. Then, in an incremental way, the modeler can define new behaviors using composition expressions. These expressions are in fact used to construct the intended behavioral model instead of explicitly describing it. The evaluation of a composition expression leads to the construction of a new behavior, which results from the composition of two existing use cases. Each composition expression specifies the use cases to be composed. We call them the base and the referred use case, respectively. The base use case represents the location where the new behavior will be added, and the referred use case represents the additional behavior to be inserted within the base use case behavior. Moreover, these expressions provide the composition operators that define the semantics of the behavioral merging, as well as the extension points where the insertion will be performed. As shown in Figure 34, the composition engine will automatically generate the new behavior.

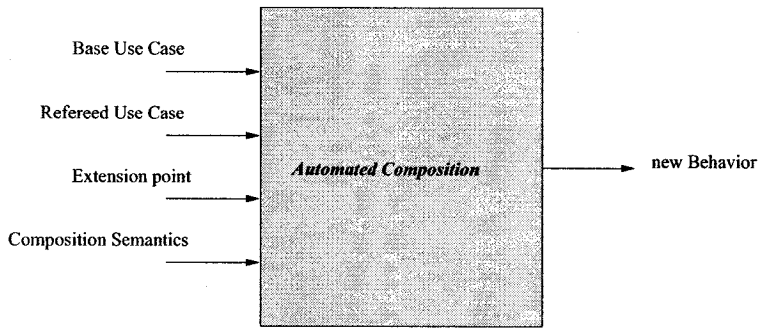


Figure 34: Composition Approach Overview

6.2.1 Incremental Process Definition

In an incremental process, the system specification (a model representing the overall set of requirements) gradually increases over iterations. The idea is to start from a set of elementary use cases to generate, increment by increment, "bigger" use cases, representing less partial behaviors. The process ends when all system requirements have been addressed in a single use case that represents the complete system functionality.

The process starts by defining the set of elementary use cases, each represents ideally a unique functionality. Then, new relationships between use cases are defined through the specification of composition expressions. When these expressions are evaluated, we add resulting use cases to the set of elementary ones. The obtained set is in fact the set of use cases that can be used in the next increment of the system specification generation.

The composition expressions are used not only to define new use cases but also to modify existing ones. This is achieved by adding a new behavior to an existing use case (either as sequencing or as alternative to a certain action), or by refining an existing use case with another use case.

Beside using expressions to define or modify use cases, the modeler can decide to remove an existing use case, or to add a modification to a use case by modifying its structure (add

a scenario to a use case, remove a certain action from a use case ...). Such modifications have an impact on the overall specification. In fact, when modifying a use case, all the constructed use cases (defined by composition expressions) have to be revised according to this change. Hence, tracking back these changes in a consistent manner is required in order to revise the system specification. As we will see in chapter 9, the composition expression order will play an important role in use case traceability.

We consider the modification of an internal structure of a use case as an incremental modification because it reflects the change of a requirement over time. When a use case is modified, all its depending use cases will be re-constructed to reflect this modification in the same increment of the specification. The next increment will be using the newly generated use cases.

6.2.2 Approach Assumptions

The synthesis of a formal specification of the system starts by specifying a certain number of elementary use cases. We do the assumption that the elementary use cases are correct and valid, and hence no verification and validation tasks are required for them.

However, when starting composing use cases, the generated use cases (by composition expression evaluation) may not respond to the needs of the modeler. Hence, a validation and verification task may be necessary before forwarding in the system specification synthesis. The V&V task can be performed in each iteration of the system specification, according to the needs of the modeler. For this reason, we choose to link our tool of use case composition to a model checker. In the case of non conformance to the modeler needs, two solutions are possible: either the use cases may be changed or the composition expression is revised.

Let's consider the example of specification built in Figure 35. When starting building the

system specification, the modeler defined two elementary use cases A and B. S/he defined two other use cases Y and Z by composition expression. These two use cases are built in the second increment and added to the original use cases A and B. Use case Y and Z may need a validation and verification task. The second increment defines the use case C, D and E using use cases A, B, Y, and Z. We note that in the third increment, no new use case is defined by composition expression. The use case D has been modified using a composition expression, evaluated in the fourth increment. The set of use cases from the third to the fourth increment did not change.

The second assumption we consider is that the set of elementary use cases are not overlapping. Indeed, each use case represents a certain requirement. They do not represent the same requirement with different views, a possible option in the case of chapter 4. Hence, with this approach, we cannot do a view consolidation. If two use cases represent an overlapping, two possibilities are to consider: either the two behaviors will still considered independent according to our mechanism of composition; or, similarly to the approach in [33], the modeler has to merge them in a single one before starting the construction of the system state based specification. Such merger can be achieved using a synchronization-like mechanism on the overlapping part, as we will show in the next chapter.

6.3 Composition Expression

For the system to work, we have to compose use cases into a consistent whole in order to validate and verify its behavior. We need to define a mechanism for composing use cases according to the behavioral weaving the modeler needs. The idea is to find a way where the merging could occur between any two use cases of the specification according to some

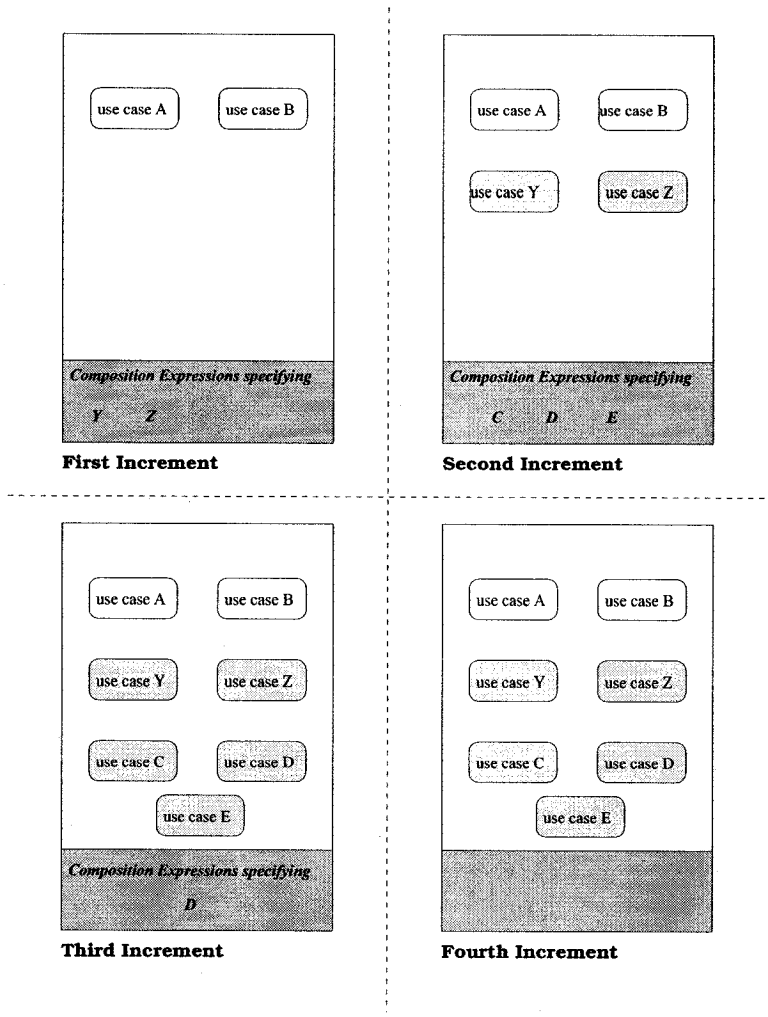


Figure 35: Different Increments in Building a Specification

semantics given by the modeler. To achieve this goal, we propose the notion of *composition expressions.* A composition expression is an expression that defines a new behavior resulting from the composition of two existing ones according to a composition operator.

The main advantages of theses expressions are :

1. the iterative aspect of the composition, which means that rather than specifying the possible extensions when describing the use case itself, an extension can be specified independently.

2. the non classification of use cases into different sets as mentioned in other works [48]. In fact, a use case can play different roles in different expressions. It can be a referred use case in an expression and a base use case in another, a fact that gives more freedom to the modeler in order to synthesize the intented whole behavior.

The syntax of the composition expression will differ according to the use case state based model being used.

6.3.1 Operator Definition

Operators define templates of use cases'composition. They allow the derivation of a new behavior from two existing ones. We have identified six operators: *Include*, *Extend_with*, *Alternative*, *Graft*, *Refine*, and *Interrupt*.

Include Operator

With the *Include* operator, the resulting use case is composed from the behavior of the base use case where we insert the behavior of the referred use case at the extension point. With this operator, some traces of the base use case may be modified. These traces represent the set of traces that pass by the extension points and where the traces of the referred use case are inserted. Fig. 6.3.1(b) shows the expected behavior from composing two use cases A and B with the operator *Include*. We note that the inclusion of the referred behavior in the base behavior is mandatory.

Extend_with Operator

In the case of the *Extend_with* operator, the resulting use case is composed of the behavior of the base use case and the behavior of the base use case where we have inserted the behavior

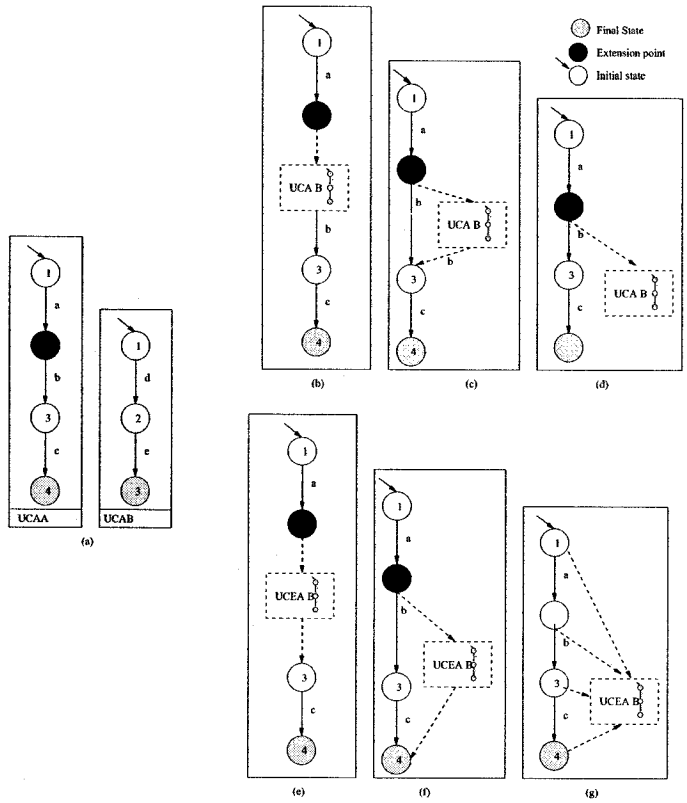


Figure 36: Expected Behavior from Composing Use Cases with the Different Composition Operators.
(a) the original use cases A and B. (b) The expected behavior with *Include* operator. (c) The expected behavior with *Extend_with* operator. (d) The expected behavior with *Alternative* operator. (e) The expected behavior with $Refine$ operator. (f) The expected behavior with $Graft$ operator. (g) The expected behavior with *Interrupt* operator.

of the referred use case in the extension points. With this operator, all of the traces of the base use case are kept and augmented by the set of traces obtained by inserting the referred use case traces into the base use case traces passing by the extension points. Fig. 6.3.1(c) shows the expected behavior from composing two use cases A and B with the operator *Extend_with*. We note that the two traces $a.b.c$ and $a.tr(B).b.c$, where $tr(B)$ is a trace of the use case B, are two possible behavior of the new generated use case. Hence, the inclusion of the referred behavior in the base behavior can be qualified as optional.

Alternative Operator

In the case of the *Alternative* operator, the resulting use case is composed of the behavior of the base use case and the behavior of the referred use case as an alternative behavior in the extension points. This operator could be called *Branching*. With this operator, all the traces of the base use case are kept and augmented by the set of traces where the traces of the referred use case are inserted into the traces of the base use case in the extension points. Fig. 6.3.1(d) shows the expected behavior from composing two use cases A and B with the operator *Alternative*. We note that the inclusion of the referred behavior in the base behavior in this case is optional. Unlike the other operators, the base use case behavior cannot resume from the extension points.

Refine Operator

Refine is an operator used to refine a transition. In other term, when using *Refine*, a transition is replaced by a use case in the resulting new behavior. The starting point of the use case is the outgoing state of the transition. The ending point of the use case is the ingoing state of the transition to be replaced. This operator is used with transition extension point. We draw the attention that refining before the transition or after the transition leads to the same behavior since we remove the transition being refined.

Graft Operator

Graft operator is used in order to add an alternative behavior to a segment of transitions. Contrarily to the previously presented operators, *Graft* needs the specification of an extension and an ending point. The extension point is where the new behavior would start and the ending point is where it finishes. If the extension and the ending points are the same,

Graft will have the same semantics than *Extend_with*.

Interrupt **Operator**

This operator is equivalent to having an *Alternative* where the extension points are all the states of the base use case. It is used when the modeler specifies a possible interrupting event from anywhere in the base use case. Figure 6.3.1 (e), (f), and (g) shows the semantics of *Refine*, *Graft*, and *Interrupt* respectively.

This thesis presents our approach considering these operators only. However, our approach is not limited to them. The modeler can easily define new operators.

We draw the attention to the fact that *Extend_with*, *Include* and *Alternative* are similar to the ones previously defined in chapter 5. However, they are no more defined as interactions, specified with conditions of the user.

6.3.2 Extension Point Definition

Extension point is a state or transition of the base use case. In order to accommodate all the operators we have defined, the syntax of extension point has to handle states, transitions, couples, and an ALL qualifier. Hence, the syntax will be:

$$
\begin{aligned}
Extension\_Points ::= \quad & IN\ State \\
& \mid qualifier\ Transition \\
& \mid IN(\ State,\ State) \\
& \mid (qualifier Transition,\ qualifier Transition) \\
& \mid ALL
\end{aligned}
$$

$$
qualifier ::= \quad BEFORE \mid AFTER
$$

When the extension points are transitions, the qualifiers *BEFORE* and *AFTER* are necessary to identify unambiguously the point where the composition is to be performed. This is not applied when the extension points are states.

The couple is used with $Graft$ operator. As said previously, the semantics of $Graft$ is the same as *Extend_with* when the elements of the couple are identical. In the implementation of our approach, the tool verifies automatically that the two elements of the couple are different, otherwise it will mention it to the designer. The *ALL* qualifier is used in order to specify that the set of extension points is in fact all the states of the use case.

Extension point Query with model checking

Rather than pointing out the extension points, it is possible to determine them automatically using a model checker. When the modeler specifies the composition expression, s/he may define a property that should be verified by the extension points. More formally, in this case the extension points will be defined as the set of states of the base use case that verify a property P.

After the definition of the composition expression, the property as well as the base use case is sent for model checking. A kripke structure [59] is generated from the use case as shown in Figure 37. As stated before, this property is used to find the set of states on which the composition should be performed. Since model checkers return only true or false with a counterexample, a way to detect the states of the base use case where the property is verified has to be developed. As we know, a positive answer from the model checker means that the property is verified in the initial state. Hence, for each state of the base use case, we run the model checker as if it was the initial state of the base use case. If it returns

true then the property holds in that state, if it returns a counterexample, then the property does not hold in that state and is not a member of our extension point set. The resulting set of states would act as the places where the composition should be done.

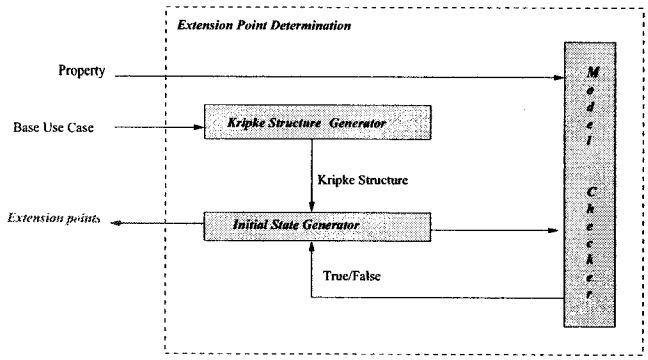


Figure 37: Extension Point Query with Model Checking

As a result of the model checking step, the extension point set could be empty or not. In case of empty set, the base use case will never verify such property and no new use case can be generated from the evaluation of the composition expression. Therefore a revision of either the property or the use cases is needed. On the other hand, when the resulting extension point set contains more than one state, the composition of the two use cases should be done in all these states.

6.4 Summary

The motivation of this approach comes from the need to formalize use case composition along with making easier the process of generating a formal model of the intended system behavior. We advocate an automated and incremental approach for the generation of such model. Incremental generation is a key issue in use case approaches. The fact that use

cases are used early in the lifecycle process requires an approach where modifications can be made easily. Using iterations is very helpful.

In our approach, the analyst starts by defining a set of partial behaviors. Then, using composition expressions s/he generates new ones, and so on until the overall behavioral model is generated. To accomplish this, we have introduced composition operators that have well-defined semantics. Unlike other use case composition approaches, our operators offer the opportunity to insert use cases inside others, which gives another level of granularity to the use case composition. The extension points can be states or transitions of the base use case.

In this chapter, we have described the overall view of our approach. It has as a source notation of use cases a state based model. After composition, it generates, a new behavior in the same state based model. The approach uses expressions in order to define a new behavior, which differ from previous approaches and also from the composition model we proposed in chapter 5. Use cases are kept independent from the composition specification itself, fact that helps the incremental aspect of the approach. In the next chapter, we will define the formal semantics of such approach in the case of an automaton.

Chapter 7

Formalization of the Approach in the Case of Use Case Automata

7.1 Introduction

In order to illustrate the novel approach of use case composition described in the previous chapter, we start by presenting it in the case of finite state automaton. In this chapter, we define a use case as a finite state machine we call Use Case Automaton (UCA). Our choice of state based model to represent use case is based on the importance of the generation a state based model of the system for the validation and the verification step of the user requirements before forwarding in the system lifecycle.

In this chapter, we propose to formalize our approach. Since the composition may be done in a single extension point or in a multiple extension points, we will first present the approach in the case of one extension point. Then, we will focus on the general case.

This chapter is composed as follows:

[1]Published in [77] and in [71]

- Section 7.2 presents the state based model of use cases.
- Section 7.3 presents the syntax of the composition expression as well as the formal definition of the operators in the case of UCA.
- Section 7.4 presents the composition approach in the case of a unique extension point.
- Section 7.5 presents a generalization of the composition approach in the case of multiple extension points.

7.2 Use Case Model Definition

A UCA is defined as a 5-tuple (S, s^0, S^f, L, E), where S is the set of states, $s^0 \in S$ is the initial state, $S^f \subseteq S$ is the set of final states, L is the set of labels, and $E \subseteq S \times L \times S$ is the set of transitions. For a transition $(s, l, s') \in E$, we write $s \xrightarrow{l} s'$. A scenario is a possible trace of this use case. More specifically, a scenario is a word of the language of the use case automaton that starts at the initial state of the UCA and finishes in one of its final states. Figure 38 (a) shows a use case X where x_1 is the initial state, x_3 and x_4 are the final states, and $\{a, b, c, d\}$ is the alphabet of X.

7.3 Composition Expression : Syntax and Semantics

7.3.1 Composition Expression Syntax

Composition expressions are used to specify the composition information necessary to compose two UCAs. They follow this syntax:

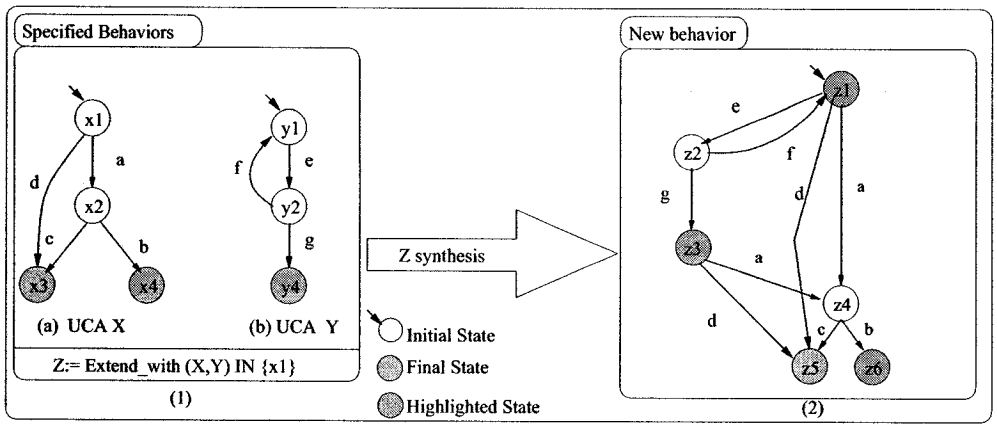


Figure 38: Expected Result from Composition

$$Z := Composition\_Operator(X, Y)Extension\_Point$$

where *Composition_Operator* is one of the composition operators. X is the base use case where to add the new behavior. Y is the referred use case (the behavioral to add to the base one). Z is the resulting UCA from the composition. As said previously, when the extension point is a state, the qualifier Before and After are not needed. In Appendix A, we prove that BEFORE a state and AFTER a state lead to a trace equivalent automata.

Figure 38 shows an example of a given user model behavior: the two UCAs, X and Y, as well as a composition expression. The new UCA Z results from the extension of the base UCA X by the referred UCA Y in the extension point x_1. UCA Z is expected to be a trace equivalent with the automaton in Figure 38 (b). Since we aim to insert complete traces of the use case Y in the extension point, this implies that during the composition, at the level of the initial and the final states of the referred use case, we have to somehow guarantee that when the execution of the referred use case starts, it continues until reaching one of its final states. Traces such as $e.f.a.c$ are considered as unexpected behaviors and we have to avoid them. This is shown in Figure 38 (2) by highlighting the two states z_1 and z_3.

7.3.2 Formal Definition of the Composition Operators

Before giving the formal details of the composition approach, we propose to present a formal definition of the composition operators in the case of use case automaton. For that purpose, we will consider the case of a unique extension point.

We use operational semantics for such definitions. The equation

$$\frac{A\ ;\ B}{C}(Cond) \tag{1}$$

means that A and B implies C. $Cond$ is the condition of the applicability of the rule.

Let's formally specify the semantics of the composition in an extension point ep. Let $A = (q_1, s_1^0, S_1^f, L_1, E_1)$ be the base use case and $B = (q_2, s_2^0, S_2^f, L_2, E_2)$ the referred use case. $C = (q, s^0, S^f, L, E)$ is the use case resulting from their composition.

Definition 9. *Let $A = (q, s^0, S^f, L, E)$ be a UCA. An execution of A is a sequence of transitions $e = q_0.q_1.q_2....q_{n-1}$ where $q_i = (s_i, l_i, s_{i+1})$, $s_i \in Q$, and $l_i \in L$ such that $s_0 = s^0$, $\forall i,\ 0 \leq i < n-1$, $(s_i, l_i, s_{i+1}) \in E$, and $q_{n-1} = (s_{n-1}, l_n, s_n) \in E$ where $s_n \in S^f$.*

Let $ex(A) = \{e | e$ *is an execution of* $A\}$ as the set of executions of the UCA. We define the set $ex(A, ep)$ be the set of executions of A passing by the extension point ep. In order to define this set formally, we have to distinguish the case of state extension point and the case of transition extension point. In the case of state extension point,

$ex(A, ep) = \{e \in ex(A) | e = q_0.q_1.q_2...q_{n-1} | \exists (ep, l_i, s_{i+1})$ *or* $q_i = (s_i, l_i, ep)\}$.

It represents the set of executions where the state ep appears as an ingoing state of a transition, or an outgoing state of a transition. In the case of transition extension point,

$ex(A, ep) = \{e \in ex(A) | e = q_0.q_1.q_2...q_{n-1} | \exists q_i = ep\}$.

It represents the set of executions where ep appears as a transition in the execution.

Finally, we define the set of prefixes $Pref(A, ep)$ (respectively postfixes $Post(A, ep)$) as the set of prefixes (respectively postfixes) of the executions of the UCA A passing by the extension point ep. More formally, let $e = q_0.q_1.q_2...q_{n-1}$ an execution of UCA A.

$$pref \in Pref(A, ep) = \begin{cases} pref = q_0.q_1....q_i & if\ (ep \in S\ and\ q_i = (s_i, l_i, ep)) \\ pref = q_0.q_1....q_{i-1} & if\ (ep \in S\ and\ q_i = (ep, l_i, s_{i+1})) \\ pref = q_0.q_1....q_{i-1} & if\ (ep \in T\ and\ q_i = ep) \end{cases}$$

and

$$post \in Post(A, ep) = \begin{cases} post = q_{i+1}.....q_{n-1} & if\ (ep \in S\ and\ q_i = (s_i, l_i, ep)) \\ post = q_i.....q_{n-1} & if\ (ep \in S\ and\ q_i = (ep, l_i, s_{i+1})) \\ post = q_i.....q_{n-1} & if\ (ep \in T\ and\ q_i = ep) \end{cases}$$

Consequently, an execution $e \in ex(A, ep)$ can be written as : $e = u.r$ where $u \in Pref(A, ep)$ and $r \in Post(A, ep)$. Let $ex(A)$ and $ex(B)$ the executions of the UCA A and B respectively. The following rules define the set of executions of the UCA C, $ex(C)$, in the case of each operator.

- **Case of $Include(A, B)$ and state extension point**

$$\frac{e \in ex(A) \setminus ex(A, ep)}{e \in ex(C)} \quad (2)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.r \ where \ u \in Pref(A, ep), \ r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.e_b.r \in ex(C)} \quad (3)$$

The first equation shows that all the executions not passing by the extension point remain unchanged. The second equation shows that the executions passing by the extension point in the UCA A are extended with the executions of the UCA B in the extension point and results in an execution of the UCA C.

- **Case of $Extend\_with(A, B)$ and state extension point**

$$\frac{e \in ex(A)}{e \in ex(C)} \quad (4)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.r \ where \ u \in Pref(A, ep), \ r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.e_b.r \in ex(C)} \quad (5)$$

The first equation shows that all the executions (passing or not by the extension point) remain executions of the UCA C. The second equation shows that the executions passing by the extension point in the UCA A are extended with the executions of the UCA B in the extension point and results in an execution of the UCA C.

- **Case of $Alternative(A, B)$ and state extension point**

$$\frac{e \in ex(A)}{e \in ex(C)} \quad (6)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.r \ where \ u \in Pref(A, ep), \ r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.e_b \in ex(C)} \quad (7)$$

The first equation shows that all the executions (passing or not by the extension point) remain executions of the UCA C. The second equation shows that only the prefixes of the executions passing by the extension point in the UCA A are kept and they are augmented with the executions of the UCA B in the extension point to form executions of the UCA C. $Alternative(A, B)$ in an extension point s can be interpreted as the possible interruption of the execution of A when reaching the state s and the start of the execution of B.

- **Case of $Refine(A, B)$ and transition extension point $t = (s, a, s')$**

$$\frac{e \in ex(A) \setminus ex(A, ep)}{e \in ex(C)} \quad (8)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.t.r \ where \ u \in Pref(A, t), \ t.r \in Post(A, t));$$

$$\frac{(e_b \in ex(B))}{u.e_b.r \in ex(C)} \quad (9)$$

The first equation shows that all the executions not passing by the extension point

transition remain executions of the UCA C. The second equation shows that t is replaced by an execution of the use case B in all the executions passing by the extension point.

- **Case of** $Interrupt(A, B)$

$$\frac{e \in ex(A)}{e \in ex(C)} \quad (10)$$

$$(e \in ex(A));$$

$$(\forall s \in S_1, \forall e = u.r \; where \; u \in Pref(A, s), \; r \in Post(A, s));$$

$$\frac{(e_b \in ex(B))}{u.e_b \in ex(C)} \quad (11)$$

The first equation shows that all the executions (passing or not by the extension point) remain executions of the UCA C. The second equation shows that all the prefixes of the executions passing by each state in the UCA A are kept and they are augmented with the executions of the UCA B to form executions of the UCA C. $Interrupt(A, B)$ can be interpreted as the possible interruption of the execution of A in any state s and the start of the execution of B.

- **Case of** $Graft(A, B)IN(s_1, s_2)$

$$\frac{e \in ex(A)}{e \in ex(C)} \quad (12)$$

$$(e_1 \in ex(A, s_1) | \exists u_1,\ r_1 \mid e_1 = u_1.r_1\ where\ u_1 \in Pref(A, s_1),\ r_1 \in Post(A, s_1));$$

$$(e_2 \in ex(A, s_2) | \exists u_2,\ r_2 \mid e_2 = u_2.r_2\ where\ u_2 \in Pref(A, s_2),\ r_2 \in Post(A, s_2));$$

$$\frac{(e_b \in ex(B))}{u_1.e_b.r_2 \in ex(C)} \quad (13)$$

The first equation shows that all the executions (passing or not by the extension points) remain executions of the UCA C. The second equation shows that the new executions of the use case C, that are not executions of the use case A, are formed by inserting an execution of B between the two extension points. In fact, this rule covers the cases where the two extension points are connected (the execution of use case A is passing by both extension points), and not connected.

The rest of the formal definition of the operators with transition extension point are presented in Appendix B.

7.4 Composition of UCAs in the Case of a Unique Extension Point

In order to compose UCAs, we advocate an automated and formal mechanism. For this purpose, we define the notion of label matching composition. The label matching composition is a composition mechanism of 2 UCAs. Such mechanism will be used later in order to build the UCA resulting from the evaluation of a composition expression.

Definition 10. *We define the label matching based composition of two UCAs* $U_1 = (S_1, s^o{}_1, S^f_1, L_1, E_1)$ *and* $U_2 = (S_2, s^o{}_2, S^f_2, L_2, E_2)$ *as a UCA* $U = (S, s^0, S^f, L, E)$ *such that :*

- $S \subseteq S_1 \times S_2$ *such that all the states of* S *are reachable from* s^o,
- $s^o = (s^o{}_1, s^o{}_2) \in S$,
- $S^f \subseteq (S^f{}_1 \times S_2) \cup (S_1 \times S^f{}_2)$
- $L \subseteq (L_1 \cup L_2)$,
- $E \subseteq S \times L \times S$, *where* E *and* S *are inferred by the following rules:*

$$\frac{((s_1, s_2) \in S);\ (s_1 \overset{l \in (L_1 \setminus L_2)}{\longrightarrow} s'_1 \in E_1)}{((s_1, s_2) \overset{l}{\longrightarrow} (s'_1, s_2) \in E);\ ((s'_1, s_2) \in S)} \tag{14}$$

$$\frac{((s_1, s_2) \in S);\ (s_2 \overset{l \in (L_2 \setminus L_1)}{\longrightarrow} s'_2 \in E_2)}{((s_1, s_2) \overset{l}{\longrightarrow} (s_1, s'_2) \in E);\ ((s_1, s'_2) \in S)} \tag{15}$$

$$\frac{((s_1, s_2) \in S);\ (s_1 \overset{l \in (L_1 \cap L_2)}{\longrightarrow} s'_1 \in E_1);\ (s_2 \overset{l \in (L_1 \cap L_2)}{\longrightarrow} s'_2 \in E_2)}{((s_1, s_2) \overset{l}{\longrightarrow} (s'_1, s'_2) \in E);\ ((s'_1, s'_2) \in S)} \tag{16}$$

Rule 14 and Rule 15 state that when a label belongs to only one UCA, then the transition tagged with this label can be fired. Rule 16 shows that when a label belongs to two UCAs, these UCAs synchronize to fire simultaneously the transition tagged with this label.

It is important to note that the label matching based composition is very similar to the well known synchronized parallel composition of labeled transition systems (LTS) [15]. However, our purpose is different from LTS since we aim at merging a UCA with another one, as it will be presented later. We are dealing with sequential system only.

In addition, the final states of the obtained automata are not necessarily the composite states of the final states of the original use cases, as it is the case in synchronized parallel composition. In our case, they represent a subset of the set of the composite states that contain at least one final state of the base and/or referred UCAs, as mentioned in the

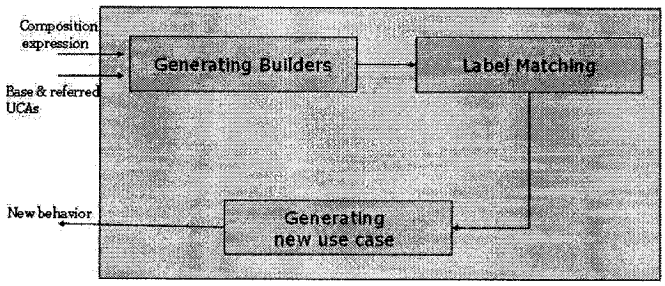


Figure 39: Approach Description in the Case of a Unique Extension Point

definition. This subset will be determined according to the semantics of the composition operator used and the extension point, as we will see later on. The best illustrative example of our decision is the sequential concatenation of two UCAs A and B, where A is the base UCA and B is the referred one. The final states of the generated UCA would be the final states of the referred UCA B. The final states of A are no longer the final states of the generated overall behavior.

7.4.1 Concept Description

In order to evaluate a composition expression, we propose to derive from the referred use case and the base use case state based models that handle the semantics of the composition expression in the extension point. These state based models are called *builders*. They reflect in fact the semantics of the composition operators in the extension point. The label matching mechanism is then used in order to compose these builders and generate the evaluation behavior of the composition expression (c.f. Figure 39).

Let's consider the case of the use case A and B in figure 40. The designer wants to include the behavior of the use case B in the use case A after the first transition a. S/he is expecting to obtain after composition the UCA that exhibits the trace $a.x.y.b$.

To derive such behavior automatically, we first derive the builders as shown in figure

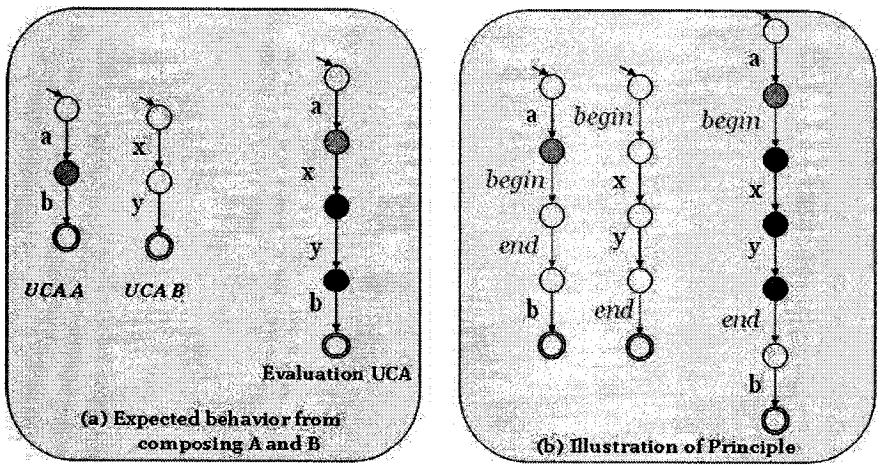


Figure 40: Illustration of the Composition Concept

40(b). On the one hand, two transitions have been added after the transition *a*, *begin* and *end*. These transitions reflect the starting point of the insertion of the behavior of the use case *B* and the ending point of this insertion, respectively. On the other hand, a transition is added at the beginning of the use case *B* labeled with *begin* and a transition is added at the end of the UCA *B* labeled with end. *begin* and *end* play the role of a synchronization labels between the two builders when applying the label matching. The builder generated from the base use case is called *base builder* while the builder generated from the referred use case is called *referred builder.*

Since we aim to generate automatically the evaluation UCA of a composition expression, we propose to automate the generation of the base and referred builders. In what follows, we present the synthesis rules of the builders in the case of the different composition operators.

7.4.2 Base Builder Generation

From a behavioral point of view, use case composition implies that the traces of the referred UCA are inserted within the trace of the base use case in the extension point with respect

to the semantics of the composition operator of the composition expression.

As we may note, it is the structure of the base builder that changes in function of the extension point and the semantics of the operator. The referred builder is in fact independent from the composition expression. Since the synthesis rules of base builders depend on the operator as well as the extension point, *Include*, *Extend_with* and *Alternative* require the distinction of nine templates. They are generated depending on the qualifier used with the operator and the type of extension point ("before a transition", "after a transition", and "in a state"). We define a set of synthesis rules for each. $Refine$ operator is only applied in the case of transition extension point. $Refine$ before a transition or after a transition is equivalent since the transition itself is removed. *Interrupt* is defined for all the states of the base use case. $Graft$ builders are defined in the case of states and transitions.

Let's consider the synthesis rules of some builders:

- Base builder of *Extend_with* operator and state extension point

  Let

  $$Z := Extend\_with(X, Y)\ IN\ \{s\}$$

  be the composition expression, where s is a constant that represents the state extension point and $X = (S, s^0, S^f, L, E)$ is the base use case. The base builder $X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, T)$ generated from X is defined as follows:

$$\frac{}{\{q, q'\} = Q \setminus S} \tag{17}$$

$$\frac{}{Q^f = S^f}(s \notin S^f) \tag{18}$$

$$\frac{(\{q,q'\} = Q \setminus S); (q \xrightarrow{end} q' \in T)}{Q^f = S^f \cup \{q'\}} (s \in S^f) \tag{19}$$

$$\frac{}{q^0 = s^0} \tag{20}$$

$$\frac{(x \xrightarrow{l} x' \in E);\ (x \neq s)}{x \xrightarrow{l} x' \in T} \tag{21}$$

$$\frac{\begin{array}{r}(\{q,q'\} = Q \setminus S);\\ (\nexists q \xrightarrow{a} q' \in T \textit{ such that } a \in L \cup \{begin, end\});\\ (\nexists q' \xrightarrow{a} q \in T \textit{ such that } a \in L \cup \{begin, end\})\end{array}}{(s \xrightarrow{begin} q \in T);\ (q \xrightarrow{end} q' \in T)} \tag{22}$$

$$\frac{\begin{array}{c}(\{q,q'\} = Q \setminus S);\\ (s \xrightarrow{a} x' \in E);\\ (q \xrightarrow{end} q' \in T)\end{array}}{(s \xrightarrow{a} x' \in T); (q' \xrightarrow{a} x' \in T)} \tag{23}$$

Rule 17 defines the set of the states of the builder UCA. q and q' are new states added in order to specify the transitions labeled with *begin* and *end*. Rule 18 defines the set of final states of the generated base builder, which represents the same set of final states of the base UCA if the extension point is not one of the final states of the UCA. Rule 19 defines the set of final states of the generated base builder in the opposite case (s is a final state of the UCA). It shows that the extension point of the obtained builder as well as the added state q' belong to the set of final states. Rule 20 defines the initial state of the base builder as the initial state of the base use case. Rule 21 shows that all the transitions that are not outgoing from s are labeled with the same

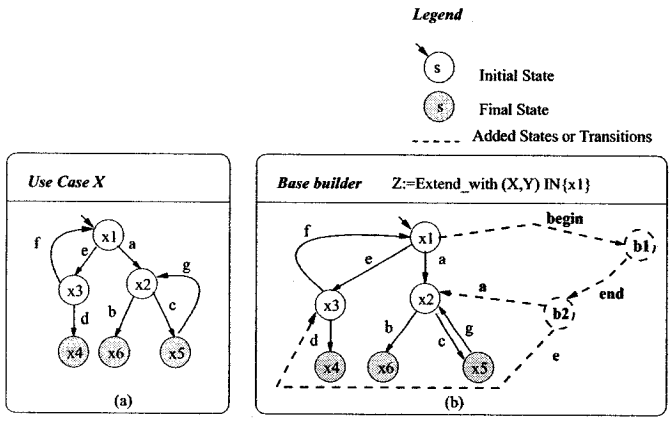


Figure 41: Base Builder for *Extend_with* Operator and State Extension Point

label in the base UCA. Rule 22 demonstrates that transitions labeled with *begin* and *end* are added in the extension point s. Finally, rule 23 shows that all the outgoing transitions of the extension point s are duplicated in order to handle the resumption of the base UCA after the insertion of the behavior of the referred UCA.

Figure 41 (b) gives an example of such a base builder generated from the UCA in Figure 41 (a). Y represents the referred UCA. The rules of synthesis of its builder are explained in the next section. x_1 is the extension point. The generation of the base builder is independent from the referred use case. Dashed lines indicate additional transitions and states we added to the structure of the base UCA in order to handle the semantics of the operator. We note that b_1 and b_2 are two additional states needed to draw transitions labeled with *begin* and *end.*

- Base builder of *Include* operator and a transition extension point with *BEFORE* qualifier

  Let

$$Z := Include(X, Y)\ BEFORE\ \{t = (s_1, a, s_2)\}$$

be the composition expression, where t is a constant that represents the transition extension point and $X = (S, s^0, S^f, L, E)$ the base UCA. The base builder $X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, T)$ generated from X is defined as follows :

$$\overline{\{q, q'\} = Q \setminus S} \tag{24}$$

$$\overline{Q^f = S^f} \tag{25}$$

$$\frac{(x \xrightarrow{l} x' \in E \; where \; (x, l, x') \neq t)}{(x \xrightarrow{l} x' \in T)} \tag{26}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \xrightarrow{a} s_2 \in E);\ (q' \xrightarrow{begin} q \notin T)}{(s_1 \xrightarrow{begin} q \in T);\ (q \xrightarrow{end} q' \in T);\ (q' \xrightarrow{a} s_2 \in T)} \tag{27}$$

Rule 24 defines the set of the states of the builder UCA. q and q' are two new states added to the base builder in order to specify the transitions labeled with $f_t(begin)$ and $f_t(end)$. Rule 25 defines the set of final states of the generated base builder. Rule 26 shows that all the transitions other than transition (s_1, a, s_2) remain unchanged in the builder. And finally Rule 27 states that the transition (s_1, a, s_2) is preceded by the new transitions labeled with *begin* and *end* that are needed to match the referred builder transitions.

Figure 42 gives an example of such a base builder generated from the UCA in Figure 41 (a). $BEFORE\ \{t = (x_1, a, x_2)\}$ is the extension point. b_1 and b_2 are the two additional states needed to draw the transitions labeled with *begin* and *end*.

- Base builder of *Alternative* operator and a transition extension point with $AFTER$ qualifier

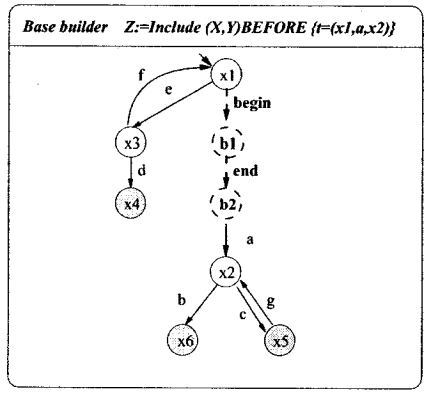


Figure 42: Base Builder in the Case of *Include* Operator and a Transition Extension Point with *BEFORE* Qualifier

Let

$$Z := Alternative(X, Y)\ AFTER\ \{t = (s_1, a, s_2)\}$$

be the composition expression where t is a constant that represents the transition extension point and $X = (S, s^0, S^f, L, E)$ is the base use case. The base builder $X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, T)$ is generated as follows:

$$\frac{}{\{q, q', q"\} = Q \setminus S} \tag{28}$$

$$\frac{(\{q, q', q"\} = Q \setminus S);\ (s_1 \xrightarrow{begin} q \in T);\ (q \xrightarrow{end} q' \in T);\ (q' \xrightarrow{a} q" \in T)}{(Q^f = S^f \cup \{q"\})} \tag{29}$$

$$\frac{(x \xrightarrow{l} x' \in E\ where\ (x, l, x') \neq t)}{(x \xrightarrow{l} x' \in T)} \tag{30}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \xrightarrow{a} s_2 \in E);\ (q' \overset{f_t(begin)}{\longrightarrow} q \notin T)}{(s_1 \xrightarrow{a} s_2 \in T);\ (s_1 \xrightarrow{a} q \in T);\ (q \xrightarrow{begin} q' \in T);\ (q' \xrightarrow{end} q" \in T)} \tag{31}$$

Rule 28 defines the set of the states of the builder UCA. q, q', and $q"$ are three new states added to the base builder in order to specify the transitions labeled with *begin* and *end*. Rule 29 defines the set of final states of the generated base builder as the set of final state of the base use case and the added state $q"$. This is due to the semantics of *Alternative* since there is no resumption to the base use case after executing the behavior of the referred one. Rule 30 shows that all the transitions other than transition (s_1, a, s_2) remain unchanged in the builder. And finally Rule 31 states that the transition (s_1, a, s_2) is followed by the new transitions labeled with *begin* and *end* that are needed to match the referred builder transitions.

Figure 43 gives an example of such a base builder generated from the UCA in Figure 41 (a). $AFTER\,\{t = (x_1, a, x_2)\}$ is the extension point. b_1, b_2, and b_3 are the additional states needed to draw the transitions labeled with *begin* and *end*.

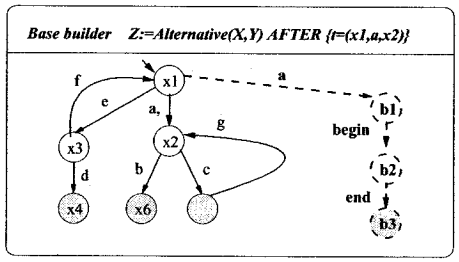


Figure 43: Base Builder in the Case of *Alternative* Operator and a Transition Extension Point with $AFTER$ Qualifier

7.4.3 Referred UCA Builder Synthesis

We recall that the synthesis of the referred builder is independent of the operator and the extension point. Each referred builder is synthesized from a referred UCA using the following rules. Let $A = (S, s^0, S^f, L, E)$ be the referred use case. The referred builder X_r is a use case UCA $X_r = (Q, q^0, Q^f, L \cup \{begin, end\}, T)$ such that:

$$\frac{}{\{q\} = Q \setminus S} \tag{32}$$

$$\frac{\{q\} = Q \setminus S}{(Q^f = \{q\});\ (s^0 = \{q\})} \tag{33}$$

$$\frac{\{q\} = Q \setminus S;}{(q \xrightarrow{begin} s^0 \in T)} \tag{34}$$

$$\frac{s \xrightarrow{l} s' \in E}{x \xrightarrow{l} x' \in T} \tag{35}$$

$$\frac{(\{q\} = Q \setminus S);\ (s \in S^f)}{(s \xrightarrow{end} q \in T)} \tag{36}$$

Rule 32 defines the set of states of the referred builder as the set of states of the referred use case increased with a new state q. Rule 33 defines the initial and the final states of the referred builder as the added state q. According to rule 34 , a transition is specified from the initial state of the builder to the corresponding state of the initial state of the referred use case clone. This transition is labeled with *begin*. Rule 35 implies that the builder evolves as the use case. Finally, rule 36 reflects that all the final states of the referred use case clone transitioned to the unique final state of the builder q with the label *end*. Figure 55 shows the referred builder generated from a referred use case the presented rules.

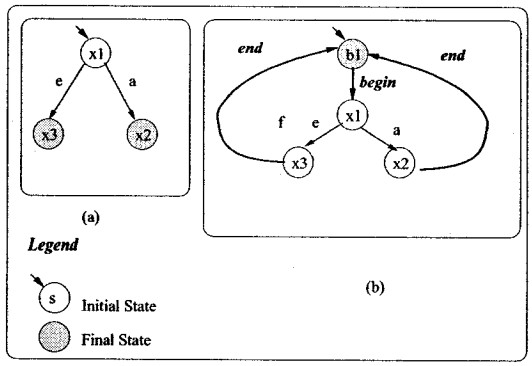


Figure 44: Referred Builder Example

7.4.4 Label Matching Composition and the evaluation UCA of the composition expression generation

When builders are generated, the label matching composition is applied. It results on a new UCA where:

- The final states are not yet defined. According to the definition of label matching, the final states are a subset of the composite states labeled with one of the final states of the base or/and referred UCA. They have to be defined more specifically according to the composition expression from where the UCA is resulting.
- Some transitions are labeled with *begin* and *end*. These transitions have to be removed because they do not belong to the set of labels of the two original UCAs.

Final State Definition

Let's consider the case of the two UCAs in Figure 45. From each use case, we have generated the corresponding builder according to the semantics of *Include* after the transition $(1, a, 2)$.

The label matching application gives the UCA in Figure 46. In the UCA, we highlighted the composite states that are labeled with one of the final states of the base or the referred

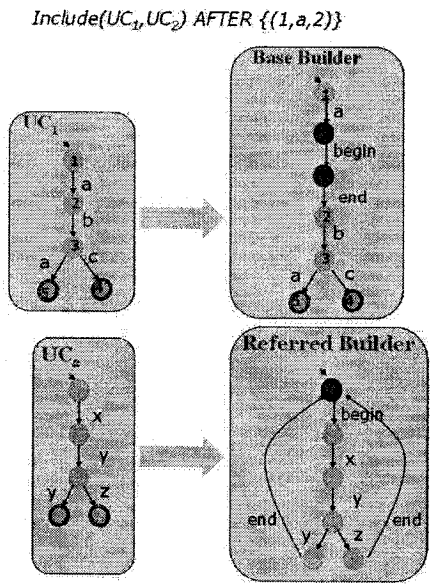


Figure 45: Builder Generations Example

use case, namely states 6, 7, 10, 11, 14, and 15.

As we may note, considering the state 6 and 7 as final states of the resulting UCA does not meet the semantics of *Include*. In fact, when including a behavior within another, the final states of the resulting behavior have to remain the same final states of the base use case. The trace $a.begin.x.y.z$ cannot be considered as a trace of the new behavior since the base behavior is not yet finished. Consequently, only the states 10, 11, 14, and 15 could be considered as final states of the new behavior. They represent the set of composite states that contain a final state of the base UCA.

Let's define the rules of determining the final states of the resulting UCA according to the composition expression.

Let S^f the set of final states of the resulting UCA from label matching $D = (S, s^0, S^f, L_1 \cup L_2 \cup \{begin, end\}, E)$. Let A_1 and A_2 be the base and the referred use cases respectively. The final states are defined with respect to the composition operator specified between A_1

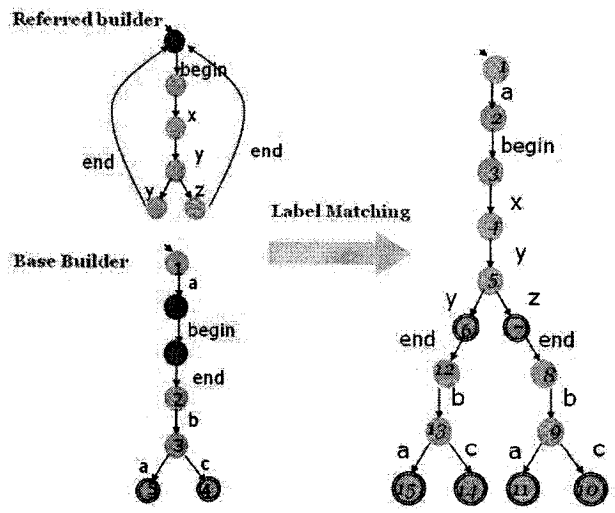


Figure 46: Label Matching Result from Builders in Figure 45

and A_2 and the extension point. We distinguish four cases:

1. *Include, Graft, and Refine* ***composition operators:***

   The set of final states of the new use case represents all the states labeled by one of the final states of the base use case, which are the final state of the base builder.

$$\frac{((s_1, s_2) \in S);\ (s_1 \in S_1^f)}{((s_1, s_2) \in S^f)} \tag{37}$$

   Figure 47 illustrates the case of *Include* operator. (a) shows the use cases to compose as well as the extension point. State 3 represent a final state and the extension point.

2. *Extend_with* ***composition operator: case of none of the final states of the base use case is an extension point:***

   The set of final states of the new use case represents all the states labeled by the final states of the base use case, which are the final states of the base builder.

$$\frac{((s_1, s_2) \in S);\ (s_1 \in S_1^f)}{((s_1, s_2) \in S^f)} (ep \notin S_1^f) \tag{38}$$

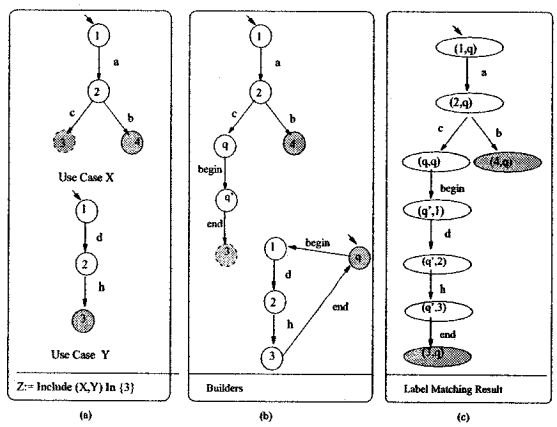


Figure 47: Example of final state Determination in the case of *Include* and one of the final states of the base use case is an extension point

3. *Extend_with* ***composition operator and the extension point is a final state of the base use case***

   This implies that the extension point is also a final state of the resulting UCA augmented by the final states of the referred UCA. This is reflected in the construction of the base builder. Consequently, the set of final states of the base use case are different from the final state of its builder. Hence, we will be using the set of final states of the generated base builder in order to determine the set of final states of the obtained UCA.

   Let $B_1 = (Q_1, q_1^0, Q_1^f, L_1, T_1)$, base builder generated from the base use case. Let $B_2 = (Q_2, q_2^0, Q_2^f, L_2, T_2)$ is the referred builder generated from the referred use case.

$$\frac{((s_1, s_2) \in S);\ (s_1 \in Q_1^f);\ (s_2 \in Q_2^f)}{((s_1, s_2) \in S^f)}(ep \in S_1^f) \tag{39}$$

The set of final states of the new use case represents all the states labeled by the final states of the base builder and final state of the referred builder.

Figure 48 illustrates such case. (a) shows the use cases to compose as well as the extension point. (b) shows the generated base and referred builders. (c) shows the intermediate use case after determining the final states.

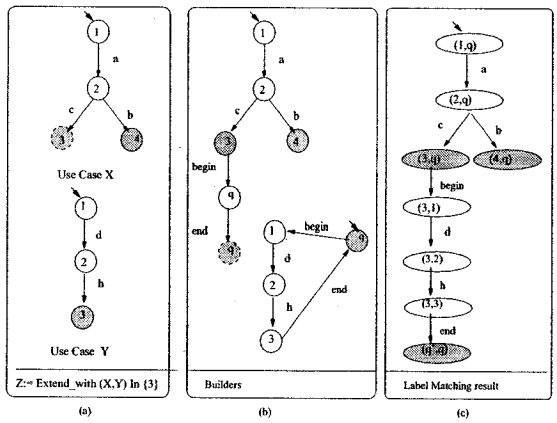


Figure 48: Final State Determination in the Case of *Extend_with* and a Final State Extension Point

4. ***When Alternative operator:***

Let $B_1 = (Q_1, q_1^0, Q_1^f, L_1, T_1)$ be the base builder and $B_2 = (Q_2, q_2^0, Q_2^f, L_2, T_2)$ be the referred builder. Therefore, the set of the final states in this case follows the rule:

$$\frac{((s_1, s_2) \in S); (s_1 \in Q_1^f \cup \{ep\}); (s_2 \in Q_2^f)}{((s_1, s_2) \in S^f)} \tag{40}$$

Figure 49 shows an example where the extension point is a final state. We notice that the final states are the composite states formed by the final states of the base use case builders and the final states of the referred use case builders $(3, q)$,$(4, q)$ and (q', q).

Figure 50 shows an example where the extension point is not a final state. We notice that the final states are those labeled by the final state of the referred use case, as it is the case of state $(2, q)$, and the states labeled by the final state of the referred and base use case builder, as it is the case of state $(3, q)$ and $(4, q)$.

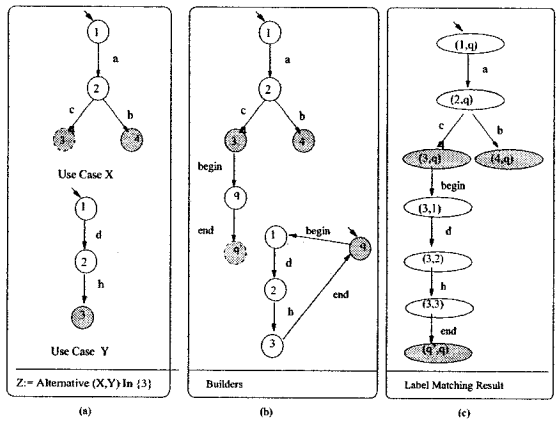


Figure 49: Case of *Alternative* Operator with a Final State Extension Point

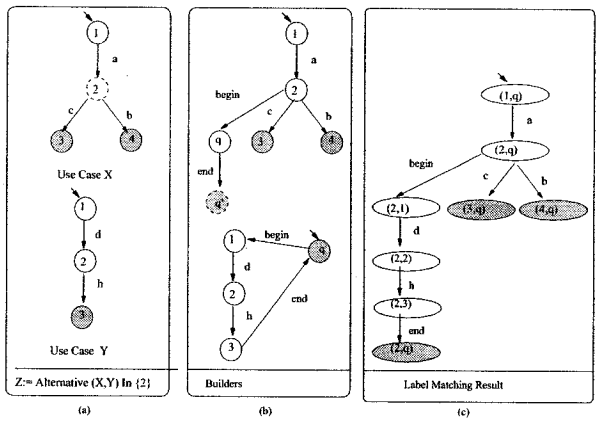


Figure 50: Case of *Alternative* Operator without a Final State Extension Point

begin and *end* Transition Removal

After determining the final states of the resulting UCA from label matching, we still need to process the UCA in order to remove the *begin* and *end* transitions. These transitions are added for composition purpose but they were not specified in the original behavior of the base and referred use cases. To do so, we propose to consider them as ϵ-transitions, and remove them with the ϵ-transition removal algorithm in [45]. The algorithm generates a trace equivalent automaton to the one obtained by label matching. This UCA is the evaluation UCA of the composition expression. Figure 51 shows the resulting algorithm

after the final state determination and the *begin* and *end* transitions removal.

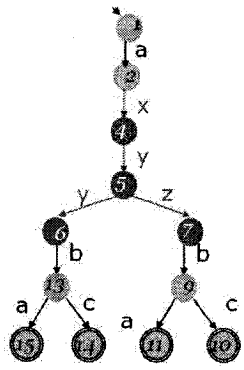


Figure 51: UCA Generated from the Label Matching in Figure 46

7.5 Generalization of the Composition Approach in the Case of Multiple Extension Points

When the extension point set is not a singleton, the composition of two UCAs, as specified by the expression, is done in each extension point (state or transition). This means inserting the referred use case in each of the extension points. Figure 52 shows the expected behavior from such composition. Since we have two extension points, state 1 and state 2, we expect to insert the behavior of the referred UCA after state 1 and after state 2, without introducing any implied scenarios. The trace *d.e.a.d.e.b* is the only accepted trace that should be generated, as shown in Figure 52 (a). In contrast, deriving the UCA in Figure 52 (b) is not expected. Despite the fact that this UCA exhibits the desired behavior *d.e.a.d.e.b*, it also introduces implied scenarios such as *d.e.a.d.e.a.d.e.b*.

In order to avoid such implied scenarios, we propose to insert a copy of the referred use case in each of the extension points. For this reason, we define the notion of *use case clone*.

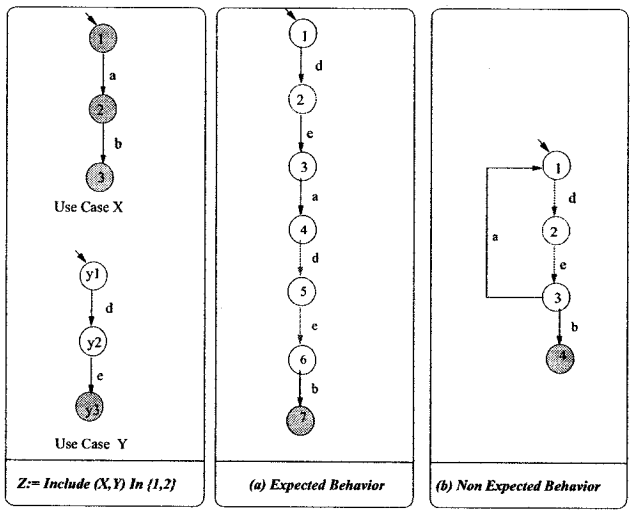


Figure 52: Expected Behavior in the Case of Multiple Extension Points

A clone of a use case is a copy of that use case that exhibits the same structure but where the transitions are relabeled according to a renaming function so that the set of labels of the use case and its clone are disjoint. Each clone is in fact inserted in a distinct extension point.

More formally, a clone of a UCA is defined as follows:

Definition 11. *Let* $X = (S, s^0, S^f, L, E)$ *a UCA. The clone of* X *with respect to a renaming function rename is a UCA* $X' = (S, s^0, S^f, L', E')$ *such that :*

- $L \cap L' = \emptyset$
- $\forall e = (s_1, l, s_2) \in E \iff \exists e' = (s_1, rename(l), s_2) \in E'$

7.5.1 Composition Description when Multiple Extension Points

In order to evaluate a UCA expression, there exists two alternatives. The first is to perform the composition in an incremental manner. This means that we first compose the two UCAs in one extension point. Then, the resulting UCA from the first iteration is used for

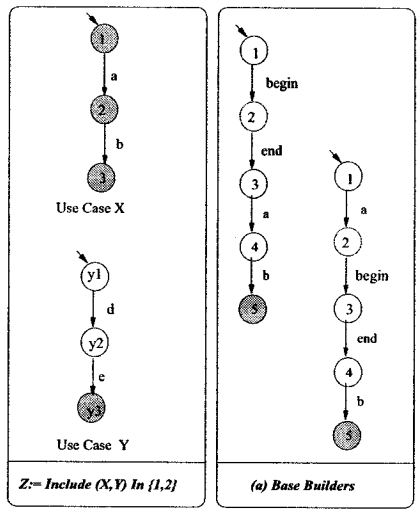


Figure 53: Base Builders in the Case of Multiple Extension Points

composition in another extension point and so on, until all the extension points have been considered. This approach leads to the problem of state traceability, since the resulting states from the first iteration are no longer the states present in the base use case. The second solution generates builders that take into account the semantics of the composition expression on the different extension points and then derive, by composition, the new behavior of the composition expression.

The fact that we have multiple extension points leads to the generation of more that two builders. Each builder reflects the semantics of the composition operator at a given extension point. Hence, as shown in Figure53, we have to generate two base builders: one in state 1 and one in state 2. In each extension point, we insert a clone of the referred use case. Hence, we end up with four builders that are composed in a further step.

In a first step, as shown in Figure 54, a set of referred use case clones are generated by a label renaming mapping procedure. Each clone will be inserted in an extension point of the base use case. Then, a set of builders is generated from both the clones of the

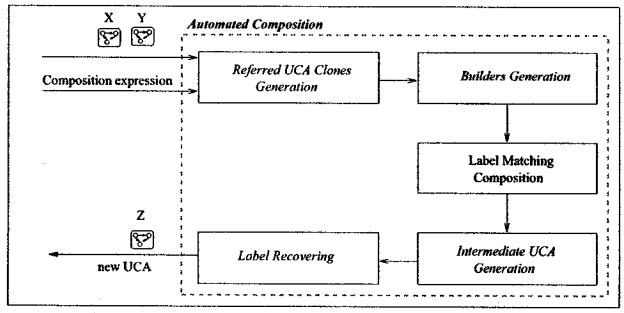


Figure 54: Composition Overview in the Case of Multiple Extension Points

referred use case and the base use case. They are synthesized automatically and handle the semantics of the operator used in the composition expression. These builders are composed, which results in a new state based model from which we extract an intermediate use case by determining the set of its final states. This set depends on the final states of the base and referred builders. The use case is qualified as intermediate because some of its label are obtained by cloning (renaming of the labels). The new behavior from the expression valuation is obtained by recovering the original labeling of the referred use case.

7.5.2 Clone Generation

Clones of a UCA are generated using a renaming function for re-labeling the transition of the original UCA. In fact, for each extension point $ep \in EP$, where EP is the set of extension points, a clone of the referred use case must be generated for two reasons: (1) to differentiate clones during the label matching and hence avoid deadlock caused by common labels, and (2) to identify the starting and the ending locations of the insertion of the referred use case clone in the extension points.

In order to automate the synthesis of the clone UCA, we define a renaming function that modifies the labeling of the UCA. It makes use of the extension point (state or transition)

where the clone will be considered for composition. Let $A_1 = (S_1, s^0{}_1, S^f_1, L_1, E_1)$ and $A_2 = (S_2, s^0{}_2, S^f{}_2, L_2, E_2)$ be two UCAs such that A_1 is the base use case and A_2 the referred one. Let EP be the set of extension points. The renaming function for an extension point $ep \in EP$ is:

$$f_{ep} : L_2 \cup \{begin, end\} \rightarrow L_2^{ep} \cup \{begin_{ep}, end_{ep}\} \tag{41}$$

$$\forall l \in L_2, f_{ep}(l) = l_{ep} \tag{42}$$

$$f_{ep}(begin) = begin_{ep} \tag{43}$$

$$f_{ep}(end) = end_{ep} \tag{44}$$

The generated clone of UCA A_2 with the renaming function f_{ep} is the UCA $A_2^{ep} = (S_2, s^0{}_2, S^f{}_2, L_2^{ep}, E_2^{ep})$. Observe that only L_2^{ep} and E_2^{ep} are not the same as in the original use case.

We have already defined the mechanism of label matching in the case of two UCAs. However, with multiple extension points, we end up with n UCAs to compose.

Definition 12. *We define the label matching based composition of n UCAs* $U_i = (S_i, s^o{}_i, S^f_i, L_i, E_i), i = 1...n$ *as an UCEA* $U = (S, s^0, S^f, L, E)$ *such that :*

- $S \subseteq S_1 \times S_2 \times ... \times S_n$ *such that all the states of* S *are reachable from* s^o,
- $s^o = (s^o{}_1, s^o{}_2, ..., s^o{}_2) \in S$,
- $S^f \subseteq (S^f{}_1 \times S_2 \times ... \times S_n) \cup (S_1 \times S^f{}_2 \times ... \times S_n) \cup (S_1 \times S_2 \times ... \times S^f{}_n)$
- $L \subseteq (L_1 \cup L_2 \cup ... \cup L_n)$,
- $E \subseteq S \times L \times S$, *where* E *and* S *are inferred by the following rule:*

$$\frac{\begin{array}{r}((s_1, s_2, ..., s_n) \in S);\\ (s_i \xrightarrow{l} s'_i \in E_i | i \in J \; where \; J = \{i \in \{1,2,..,n\} | s_i \xrightarrow{l} s'_i \in E_i\})\end{array}}{\begin{array}{c}((s_1, s_2, ..., s_n) \xrightarrow{l} (s'_1, s'_2, ..., s'n) \in E | (s'_i = s_i if i \notin J));\\ ((s'_1, s'_2, ..., s'_n) \in S)\end{array}} \quad (45)$$

Rule 45 states that when a label belongs to a single UCA, then only this UCA may fire the transition. However, when a label belongs to more than one UCA, then these UCA transitions are fired simultaneously.

7.5.3 Base UCA Builder Synthesis

For each $ep \in EP$, we construct a base builder from the use case A_1 with respect to the renaming function f_{ep}. The synthesized base builder is a UCA $A_1 = (S_1, s^0{}_1, S^f_1, L_1 \cup \{f_{ep}(begin), f_{ep}(end)\}, E^{ep}_1)$ that reflects the semantics of the composition operator as well as the extension point ep. The labels of the base use case are not renamed in the base builder, only two labels $f_{ep}(begin)$ and $f_{ep}(end)$ are added which serve as the common label indicating the start and the end of the insertion of the referred use case clone within the base one.

In Appendix C, we present three tables that cover the different synthesis rules for each composition operator. Table 6 contains the synthesis rules for the builders on extension points specified as states. Table 7 presents the synthesis rules when the $BEFORE$ qualifier is used with a transition extension point. The synthesis rules of base builders using an $AFTER$ qualifier with a transition extension point are presented in Table 8.

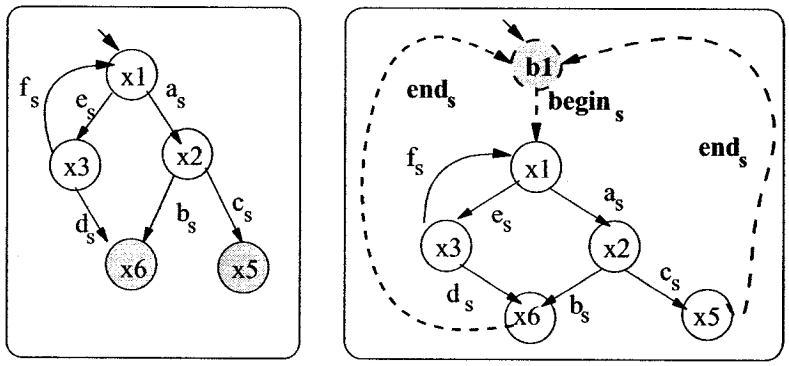


Figure 55: Example of Referred Builder

Let $A^{ep}_{2_{clone}} = (S_2, s^0{}_2, S^f{}_2, L^{ep}_2, E^{ep}_2)$ be the UCA of the referred use case clone synthesized from A_2 using the renaming function f_{ep}. The referred builder A^{ep}_2 of with the same renaming f_{ep} is a use case UCA $A^{ep}_2 = (S_2, s^0{}_2, S^f{}_2, L^{ep}_2 \cup \{f_{ep}(begin), f_{ep}(end)\}, E^{ep}_2)$ that follows the synthesis rules 32, 33, 34, 35, 36. Only the labels *begin* and *end* are replaced by $f_{ep}(begin)$ and $f_{ep}(end)$. Figure 55 (a) illustrates a clone of a referred use case with a renaming function f_s and (b) illustrates an example of a synthesized referred builder using these rules.

We note that in the case of multiple extension points, each base builder has its correspondent referred builder generated from the refereed use case clone as illustrated in Figure 56. The renaming function f_{ep} builds the link between the base and referred builders while applying the label matching composition.

7.5.4 Use Case Generation

When builders are synthesized, we apply the matching label composition to generate an intermediate UCA. During this label matching, none of the labels of the referred builders will match together because of the different labeling we impose on the clone generation step.

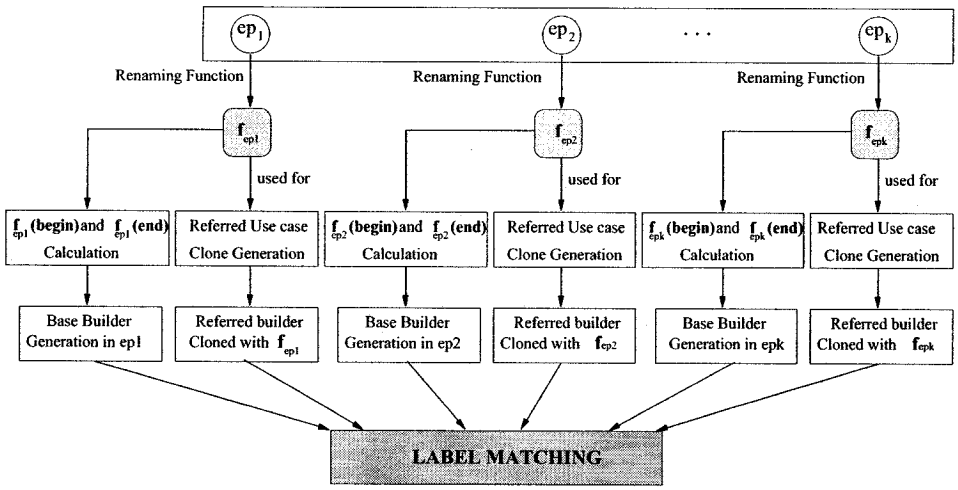


Figure 56: Composition with Multiple Extension Points

In addition, referred and base builders have to match only on $f_{ep}(begin)$ and $f_{ep}(end)$ for an extension point ep. Hence, we assume that $L_1 \cap (\cup_{\forall ep \in EP} L_{ep}) = \emptyset$.

The resulting automaton still does not represent the final use case since some of its transitions are labeled by $f_{ep}(begin)$ and $f_{ep}(end)$, where $ep \in EP$. These transitions cannot be removed by a simple state merging. This may threaten the assumption that our composition approach has no implied scenario generation (only complete traces of the referred use case are inserted in the extension point of the base use case). Hence, we consider them as $\epsilon-$transitions and we later remove them. An example of a synthesized UCA representing the UCA after ϵ-transition removal is illustrated in step (4) of Figure 57.

By recovering the original labeling of the referred use case, the final use case would be retained. Let $A_1 = (S_1, s_1^0, S_1^f, L_1, E_1)$ be the base use case and $\{A_{2_{clone}}^{ep}, ep \in EP\}$ the set of the referred use case clones where EP is the set of extension points. Let $C = (Q, q^0, Q^f, L_1 \bigcup(\bigcup_{ep \in EP} L_2^{ep}), T)$ be the intermediate generated use case. The obtained use case is intermediate since it still holds the renaming of labels used to generate the different clones of the referred use case. Consequently, we have to restore the original labeling to

obtain the final use case. For this purpose, we define a renaming function g such that:

$$g_{ep} : L_1 \cup (\cup_{ep \in EP} L_2^{ep}) \rightarrow L_1 \cup L_2 \tag{46}$$

$$\forall l \in L_1, g_{ep}(l) = l \tag{47}$$

$$\forall l \in L_2^{ep}, ep \in EP, g(f_{ep})(l) = l \tag{48}$$

The label restoration of the intermediate use case results in a new use case as shown in step (5) of Figure 57.

An example of the overall process of the composition is shown in Fig. 7. It addresses the case of composition in multiple extension points. An example of the overall process of the composition is shown in Figure 57. It addresses the case of composition with *Include* operator in multiple extension points, state 1 and 2 of the base use case. State 1 is also an initial state of the base use case. Two clones of the referred use case are generated using a renaming function f_1 and f_2 respectively. As a third step, the different builders are generated. We notice that the initial state of the first builder is different from the initial state of the base use case. This is due to the fact that the extension point is an initial state of the base use case. In the fourth step, we have applied the label matching composition as well as the final state determination. The transition removal is achieved in the fifth step. And finally, we have recovered the original labeling of the referred use case in the sixth step.

7.6 Summary

In this chapter, we have presented a formalization of our composition approach when use cases are specified as use case automata. We tackled the composition in the case of a unique

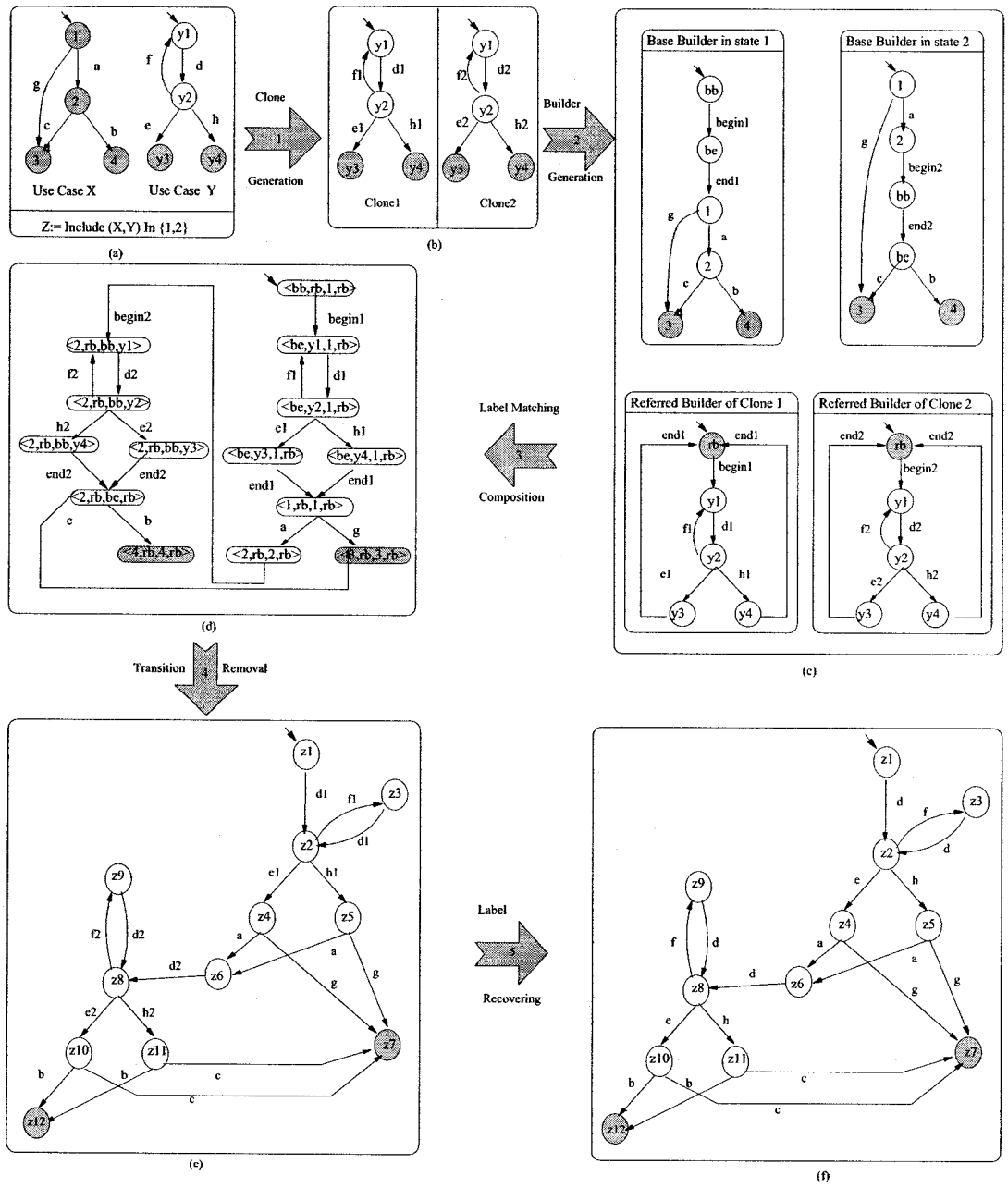


Figure 57: Example of Composition in Multiple Extension Points

extension point and of multiple extension points. For this purpose, we have presented:

- The formal definition of the use case model.
- The formal definition of the label matching algorithm, which represents the algorithm of use case composition.
- The formal definition of the composition operators in term of trace composition. And
- The formal definition of the different base builders that represent the semantics of the composition according to the operator and the extension point, and the referred builders.

In the next chapter, we present the formalization of our approach in the case of use case extended automata.

Chapter 8

Formalization of the Approach in the Case of Use Case Extended Automata

8.1 Introduction

In a conventional FSM, the transition is associated with a set of input Boolean conditions and a set of output Boolean functions. Such model is not always suitable for the specification of reactive systems, where conditions have to be verified before firing transitions. In an EFSM model, the transition can be expressed by an if statement. If trigger conditions are all satisfied, the transition is fired, performing the specified data operations and bringing the machine from the current state to the next state.

In this chapter, we apply the approach presented in chapter 6 in the case of a variant of

[1]Published in [70] and in [72]

the extended finite state machine, called Use Case Extended Automaton (UCEA). States of UCEAs are extended by variables, and transitions are guarded by pre-conditions and have assignments to update these variables.

Introducing variables allows to fold many transitions into a single transition of the UCEA. Consequently, requirements represented as UCEA would be more concise and scalable. However, it raises two issues:

1. In an ideal situation, use cases are supposed be developed independently. Introducing variables in the model of use cases implies a decision about the scope of these variables: should these variables be shared between the set of use cases or defined locally to each use case.

2. Contrarily to FSMs, a problem of executability may arise with EFSMs. It comes from the fact that when inserting a behavior in an extension point, we can not predict if the condition of the execution of that behavior will be satisfied or not.

The remainder of this chapter is organized as follows.

- Section 8.2 presents the formal definitions of the UCEA model and the label matching based composition of two UCEAs.
- Section 8.3 shows the composition expression syntax adapted to the model of UCEA.
- Section 8.4 describes the formal composition method in the case of unique extension point.
- Section 8.5 describes the composition appraoch in the case of multiple extension points.
- Section 8.6 discusses the issue of use case executability.

8.2 System Specification Model in the Case of UCEAs

In this section, we present the model of use cases. We also modify the label matching composition in order to adapt it to the model with variables.

8.2.1 Use Case Model Definition

Finite state machines are not sufficient to represent use cases in a realistic way. They represent the control part of the system behavior but they don't handle data. Hence, using such models, we may end up with non realistic representation of the system. In order to formally model the system behavior with a high level of expressiveness, we propose to extend our use case model with variables. Formally, a UCEA is defined as follows:

Definition 13. *A UCEA is a 7-tuple* $(S, s^o, S^f, L, V, I, E)$ *such that:*

- S *is the set of states,*
- s^o *is 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 pre-condition of a transition has to be true before the transition is enabled and its variable assignments play the role of a post-condition*

The set of input variables represents the variables that can be initialized when inserting the UCEA into another. A transition $(s, c, l, a, s') \in E$ is also written as $s \overset{c,l,a}{\rightarrow} s' \in E$. As defined in the chapter 4, we define C as a set of $(v_i \# c)$ where $v_i \in V, c \in dom(v_i)$, and $\#$ is a binary relation. In contrast, a is defined as a list of assignments that can be either $(v_i := v_j)$, $(v_i := v_i \ op \ const)$, or $(v_i := const)$, where v_i and v_j are variables. Once the system is in s, if the precondition c is true, the transition $s \overset{c,l,a}{\longrightarrow} s'$ is fired, and then the variables are updated by the execution of the assignments in a. The default value of any variable in V is *null* (*null* denotes the fact that at this stage, the variable is not yet initialized).

Figure 58 shows a UCEA of authentication to a bank account. We have defined three variables: *pin*, *check*, and v. *pin* is an integer, *check* is a Boolean that returns the result of the pin check, and v is an integer that counts the number of attempts to enter the pin number. At the beginning of the use case, *pin* and *check* are not initialized, while v is set to 0. The user is allowed to have 3 attempts to login to the system. s_0 is the initial state of the UCEA. s_4 and s_5 are the final states. There is no input variable specified for this use case.

As mentioned before, when extending the model with variables, decisions about the scope of these variables have to be made. Consequently, the set of variables V may contain two types of variables: *UCEA variables* and *specification variables*. The latter represent the set of shared variables between all the UCEAs that define the specification, while the former represents the set of local variables to the UCEA itself. By analogy to programming language, UCEA variables play the role of local variables while specification variables play the role of global variables to the program.

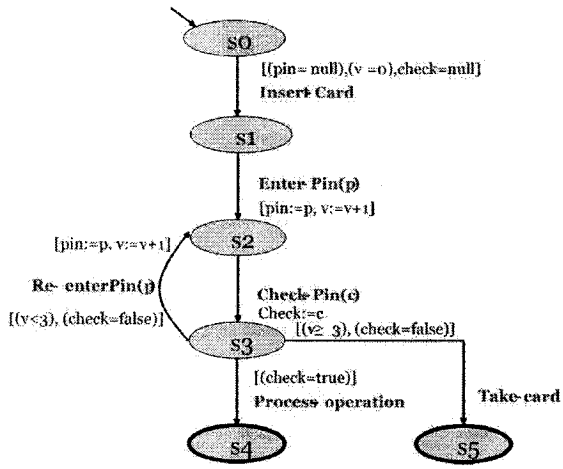


Figure 58: Example of a UCEA

We denote by $\Sigma.v$ a specification variable, where Σ is the name of the specification, and by $U.v$ the UCEA variable where U represents the name of the UCEA.

Definition 14. *A specification Σ is a couple (μ, ν) such that:*

- *μ is the set of UCEAs defined by the user (either originally or by composition)*
- *ν the set of specification variables.*

8.2.2 Label Matching Based Composition of two UCEAs

In order to handle variables information, the previous definition of label matching has to be extended. It takes into account the input variables, the set of variables, and the conditions and assignments used to decorate the use case transitions. The definition is as follows:

Definition 15. *Let $\Sigma = (\mu, \nu)$ be a specification. We define the label matching based composition of two UCEAs $U_1 = (S_1, s^o{}_1, S^f_1, L_1, V_1, I_1, E_1)$ and $U_2 = (S_2, s^o{}_2, S^f_2, L_2, V_2, I_2, E_2)$ where $U_1 \in \mu$, $U_2 \in \mu$, and $V_1 \cap V_2 \in \nu$ as an UCEA $U = (S, s^0, S^f, L, V, I, E)$ such that :*

- $S \subseteq S_1 \times S_2$ *such that all the states of* S *are reachable from* s^o *in the graph of the resulting composed UCEA,*
- $s^o = (s^o{}_1, s^o{}_2) \in S$,
- $S^f \subseteq (S^f{}_1 \times S_2) \cup (S_1 \times S^f{}_2)$
- $L \subseteq (L_1 \cup L_2)$,
- $V = (V_1 \cup V_2)$,
- $I \subseteq I_1 \cup I_2$
- $E \subseteq S \times C \times L \times A \times S$,

where E *and* S *are inferred by the following rules:*

$$\frac{((s_1, s_2) \in S);\ (s_1 \xrightarrow{c,l,a} s'_1 \in E_1);\ (s_2 \xrightarrow{c,l,a} s'_2 \in E_2)}{((s'_1, s'_2) \in S);\ ((s_1, s_2) \xrightarrow{c,l,a} (s'_1, s'_2) \in S)} \tag{49}$$

$$\frac{((s_1, s_2) \in S);\ (s_1 \xrightarrow{c,l,a} s'_1 \in E_1);\ (s_2 \xrightarrow{c,l,a} s'_2 \notin E_2)}{((s'_1, s_2) \in S);\ ((s_1, s_2) \xrightarrow{c,l,a} (s'_1, s_2) \in S)} \tag{50}$$

$$\frac{((s_1, s_2) \in S);\ (s_1 \xrightarrow{c,l,a} s'_1 \notin E_1);\ (s_2 \xrightarrow{c,l,a} s'_2 \in E_2)}{((s_1, s'_2) \in E);\ ((s_1, s_2) \xrightarrow{c,l,a} (s_1, s'_2) \in S)} \tag{51}$$

- $\Sigma = (\mu \cup U, \nu)$

Rule (49) states that when the labels, the pre-conditions and the assignments of two transitions of the UCEAs are the same, these two transitions are merged into a single transition of the resulting composed UCEA. Otherwise, as stated in rules (50) and (51), each transition is represented separately in the composed UCEA. The final states of the resulting UCEA as well as its input set of variables will be determined later on. They

depend on the role the composed UCEAs play in the composition expression, as it was the case of UCAs.

8.3 Composition Expression in the Case of UCEAs

As mentioned earlier, a UCEA composition expression specifies the way a new behavior (a UCEA) is synthesized by composing two existing UCEAs. The previously defined syntax for the composition expression is no more sufficient. Conditions on the composition may be added contrarily to the case of UCAs. In what follows, we will present the syntax of the composition expressions as well as an overview of the composition in the case of UCEAs.

8.3.1 UCEA Composition Expression Syntax

A UCEA composition expression is formed from six elements: two UCEAs, an extension point, a composition condition, and an input and output sets of assignment of variables. The composition condition is a condition that constrains the behavior of the referred use case when merging the two behaviors. Finally, since we consider that some variables are defined as local variables in the use case, it seems necessary in some cases to define a correspondence between the variables of the base and the referred use cases. This is specified within the UCEA expression in the $Input\_Var\_Assign$ and the $Output\_Var\_Assign$ fields. The UCEA composition expression follows the syntax:

$$\begin{aligned} Z ::= \; & Composition\_Operator\ (X, Y)\ Extension\_Point \\ & Composition\_Condition \\ & Input\_Var\_Assign\ Output\_Var\_Assign \end{aligned}$$

Z represents the UCEA generated from the evaluation of the composition expression. X is the base use case and Y is the referred one. $Composition\_Operator$ represents one of the

previously mentioned composition operators, *Include*, *Extend_with*, *Alternative*, *Refine*, *Graft*, and *Interrupt*.

Composition_Condition is a user specified condition used when composing the behavior of the referred use case into the base use case one. This condition follows the syntax of the pre-condition on variables when specifying the transition relations in any UCEA. It will constrain the execution of the referred use case. It may specify conditions on any variables of the base use case, and/or the system variables of the specification but not on variables of the referred use case. It is because at that point, the values of the local variables of the referred use case may be not initialized yet.

Input_Var_Assign follows this syntax:

$$
\begin{aligned}
Input\_Var\_Assign ::= \quad & [Input\_Var\_Ass\_Elt] \\
& | Input\_Var\_Ass\_Elt :: Input\_Var\_Assign \\
& | [] \\
\\
Input\_Var\_Ass\_Elt ::= \quad & (Input\_var\_Referred := var\_Base) \\
& | (Input\_var\_Referred := C) \\
& | (var\_Spec := C) \\
& | (Input\_var\_Referred := var\_Spec)
\end{aligned}
$$

It represents a list of assignments where an input variable of the referred use case is being assigned a value of one of the variables of the base use case, a constant that belongs to its domain, or the value of one of the specification variables. The two variables specified in the assignment have to be of the same type. Only the input variables of the referred UCEA can be initiated by these assignments. All the other variables are considered as local to the referred use case and remain unchanged. If no assignment is required during the

composition, $Input\_Var\_Assign$ is set to [].

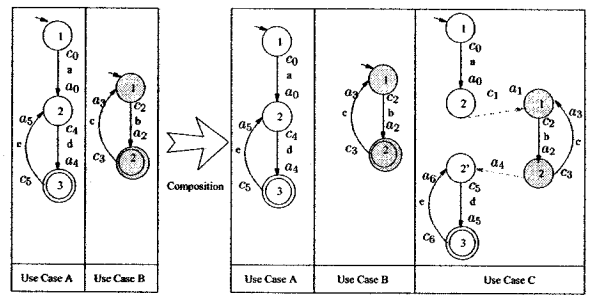
We remind that for four of the proposed composition operators, namely $Include$, $Extend\_with$, $Graft$, and $Refine$, the execution flow resumes back to the base use case. Variable assignments are, thus, also needed to update some variables of the base use case. They follow the syntax:

$$
\begin{aligned}
Output\_Var\_Assign ::= \quad & [Output\_Var\_Ass\_Elt] \\
& | Output\_Var\_Ass\_Elt :: Output\_Var\_Assign \\
& | []
\end{aligned}
$$

$$
\begin{aligned}
Output\_Var\_Ass\_Elt ::= \quad & (var\_Base := var\_Referred) \\
& | (var\_Base := C) \\
& | (var\_Spec := C) \\
& | (var\_Base := var\_Spec)
\end{aligned}
$$

The $Output\_Var\_Assign$ is a list of assignments where a variable of the base use case is being assigned a value of one of the variables of the referred use case, a constant that belongs to its domain, or to the value of one of the specification variables. The two variables specified in the assignment have to be of the same type. If no assignment is required during the composition, $Output\_Var\_Assign$ is set to [].

We note that during the specification of the composition expression, either in the input assignment or in the output assignment, the analyst may specify a new value to a specification variable. If no assignment is made, and since specification variables play the role of global variables shared between the different use cases, the referred use case is going to consider the value of the specification variable in the extension point.



(a) Specification Current Increment (b) Specification Next Increment

$$C := \quad Include(A, B)\ IN\ (2)\ [c_1]\ [a_1]\ [a_4]$$

Figure 59: System Specification Synthesis: Approach Description

8.3.2 Approach Overview

Let's consider the example in Figure 59 of composing UCEAs. The modeler starts by specifying two use cases A and B, as well as a UCEA expression C. C is a use case where the behavior of the UCEA B is included in the behavior of the UCEA A in $state2$ with the composition condition c_1, the input variable assignment a_1 and the output variable assignment a_4. The UCEA C that we aim to generate automatically is represented in Figure 59 (b). In UCEA C, c_1 and a_1 have to be considered before starting the execution of the behavior of the referred use case B. In the same way, a_4 has to be considered before resuming back to the execution of the base behavior A.

8.4 UCEA Composition Approach

As explained in the previous chapter, the composition approach consists of three steps: builders generation, label matching composition, and final states determination and ϵ-transition removal. In this section, we explain the modifications we have performed to the formal definition of these steps in order to adapt them to the UCEA model.

8.4.1 Builder Generation

For the extension point *ep*, we construct a base builder from the use case. The synthesized base builder is an UCEA that reflects the semantics of the composition operator in the extension point *ep*.

Base Builder Generation

Let $X = (S, s^0, S^f, L, V, I, E)$ be the base use case of a composition expression. We present in Appendix the synthesis rules of the base builders. Tables 9, 10, and 11 define the synthesis rules in the case of state extension point, transition extension point with the qualifiers *BEFORE*, and transition extension point with the qualifier *AFTER*, respectively. Let's consider the case of $Graft$ operator with state extension point.

$Z := Graft(X, Y)\ IN\ \{(IN\ s_1,\ IN\ s_2)\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$

where *Input_assgn* and *Out_assgn* are conditions. s_1 is a constant that represents starting point of the referred UCEA and s_2 is the ending point such that $s_1 \neq s_2$.

The base builder $X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$ is derived such that :

$$\overline{\{q\} = Q \setminus S} \tag{52}$$

$$\overline{Q^f = S^f} \tag{53}$$

$$\overline{q^0 = s^0} \tag{54}$$

$$\frac{(x \xrightarrow{c,l,a} x' \in E)}{(x \xrightarrow{c,l,a} x' \in T)} \tag{55}$$

$$\frac{(\{q\} = Q \setminus S);\ (s_2 \in s)}{(q \xrightarrow{true,end,Out\_assign} s_2 \in T)} \tag{56}$$

$$\frac{(\{q\} = Q \setminus S); (q \xrightarrow{true,end,Out\_assign} s_2 \in T);\ (s_1 \in S)}{(s_1 \xrightarrow{Cond\_comp,begin,Input\_assign} q \in T)} \tag{57}$$

Rule 52 defines the set of states of the base builder as the set of states of the base use case increased with a state q. Rule 53 defines the final state of the base builder as the final states of the base use case. Rule 54 defines the initial state of the base builder as the initial state of the base UCEA. Rule 55 implies that the builder evolves as the use case for all the transitions. According to Rule 56, a unique transition is specified between the added state q and the ending state s_2. This transition is labeled with *end* and has as post condition the *Output_assign* specified in the expression. Moreover, Rule 57 shows that a unique transition is specified between the starting state s_1 and the additional state q. This transition is labeled with *begin* and has as pre-condition the *Cond_comp* specified in the expression and as post-condition the *Input_assign* specified in the expression.

We notice that the set of variables of the generated base builder is obtained by the union of the set of variable of the base use case and the set of variables of the referred use case used in the specification of the composition condition, input assignment, and output assignment.

Figure 60 shows an example of the construction of such builder. We note that v_1, v_2, and v_3 are the set of variables of the builder. This is due to the fact that the input assignment affects the value of v_1 to v_3, which is a variable of the referred use case Y.

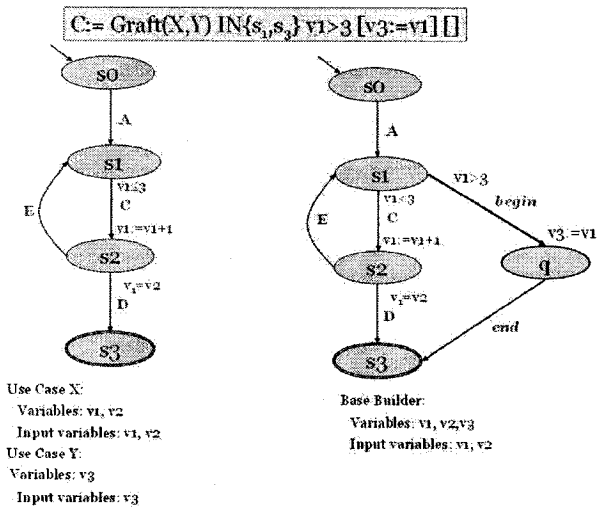


Figure 60: Base Builder in the Case of UCEA with $Graft$ Operator

Referred Builder Generation

Let $X = (S, S^0, S^f, L, V, I, E)$ the UCEA of the referred use case. The referred builder is a use case UCEA $X_r = (Q, q^0, Q^f, L \cup \{begin, end\}, V_r, I_r, T)$ [1] such that:

$$\frac{}{\{q\} = Q \setminus S} \tag{58}$$

$$\frac{(\{q\} = Q \setminus S)}{(Q^f = q);\ (q^o = q)} \tag{59}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in E)}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{60}$$

$$\frac{(\{q\} = Q \setminus S)}{(q \overset{Cond\_comp,begin,input\_assign}{\rightarrow} s^o \in T)} \tag{61}$$

$$\frac{(\{q\} = Q \setminus S);\ (x \in S^f)}{(x \overset{true,end,Out\_assign}{\rightarrow} q \in T)} \tag{62}$$

[1] *Cond_comp*, *Input_assign*, and *Out_assign* used in the synthesis rules are specified in the UCEA composition expression.

Rule 58 defines the set of states of the referred builder as the set of states of the referred use case increased with a new state q. Rule 59 defines the final state and initial state of the referred builder as the new added state q. Rule 60 implies that the builder evolves as the use case. According to Rule 61, a transition is specified from the initial state of the builder to the corresponding state of the initial state of the referred use case. This transition is labeled with *begin*. It has as a pre-condition *Cond_Comp* as specified in the expression and as a post-condition *In_assign*. Finally, Rule 62 reflects that all the final states transitioned to the unique final state q with the label *end* and the post-condition *Out_assign*.

In this case too, the set of variables of the builder is obtained by the union of the set of variable of the referred use case and the set of variables of the referred use case used in the specification of the composition condition, input assignment, and output assignment.

Figure 61 presents an example of referred builder generated from a UCEA Y. Any variable added to the set of variables of the original referred UCEA will be considered as an input variable of the referred builder. This is due to the semantics of input variables, variables that are not initialized within the use case itself. In addition, such decision will not affect the result of the composition because builders are intermediate UCEAs and do not represent the final behavior of the composition. The input variables of the resulting behavior will be decided based on the base and referred use cases, not their builders.

8.4.2 Intermediate UCEA Generation

Let $A_1 = (S_1, s_1^0, S_1^f, L_1, V_1, I_1, E_1)$ and $A_2 = (S_2, s_2^0, S_2^f, L_2, V_2, I_2, E_2)$ be the base and referred UCEAs, and $B_1 = (Q_1, q_1^0, Q_1^f, L_1 \cup \{begin, end\}, V_b, I_b, T_1)$ and $B_2 = (Q_2, q_2^0, Q_2^f, L_2 \cup \{begin, end\}, V_r, I_r, T_2)$ their respective generated builders. Let $C = (Q, q^0, Q^f, L_1 \cup L_2 \cup \{begin, end\}, V_1 \cup V_2, I_1, T)$ the UCEA obtained from the label matching based composition

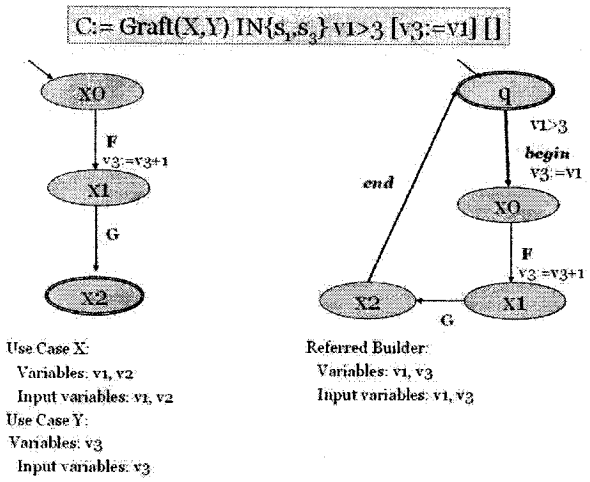


Figure 61: Example of a Referred Builder in the Case of UCEA and with $Graft$ Operator

of B_1 and B_2. Within the label matching based composition, referred and base builders have to match only on the two labels *begin* and *end* because of the assumption $L_1 \cap L_2 = \emptyset$.

The final UCEA would contain all traces of the referred use case UCEA within the base use case UCEA with regard to the specified composition operator and the conditions and assignments specified in the composition expression. The final UCEA is derived by stating the set of final states and removing the *begin* and *end* transitions.

8.4.3 Final States Determination

The determination of the final states of the UCEA obeys to the same rules we developed in the case of UCAs. This is due to the fact that variables are used as decoration to the transitions and do not intervene in the determination of such states.

8.4.4 *begin* and *end* Transitions Removal Algorithm

The obtained automaton after determining the final states still contains some $\epsilon-$transitions due to the composition mechanism we proposed. We offer to the modeler the possibility

of removing such transitions by post processing the obtained UCEA. We cannot use the same algorithm used in the case of UCAs. This is due to the fact that ϵ-transitions are now decorated with conditions and assignments that have to be taken into consideration before removing the transitions.

We propose a two step approach. In the first step, we remove the pre-conditions and the post-conditions of the $\epsilon-$transitions by propagating them on some other transitions. In the second step, we propose to apply on the derived UCEA (with non constrained $\epsilon-$transitions) the ϵ-transitions removing algorithm in [45].

Within the first step, we start by propagating forward the pre-condition of the $\epsilon-$transitions in the transitions that are outgoing from the ingoing state of the $\epsilon-$transitions. In contrast, the propagation of the post-condition is done backward, in the ingoing transitions of the outgoing state of the $\epsilon-$transitions. To do so without harming the data flow of the UCEA, we add some states to the UCEA as shown in Figure 62 in order to avoid over-constraining the obtained behavior. Let's consider the case of propagating c_1. c_1 should be verified only the first time when firing the transition $s_4 \xrightarrow{c_2,b,a_2} s_5$, but not after looping and returning back to the state s_4. For this reason, we added the state s_{41} and the transition $s_{41} \xrightarrow{c_2 \wedge c_1,b,a_2} s_5$.

In the same way, in order to propagate a_1 in transitions $s_1 \xrightarrow{c_0,a,a_0} s_2$ and $s_3 \xrightarrow{c_5,e,a_5} s_2$, we have to take into account that the assignment a_1 should be done only in the case of traces passing by state s_{41}. Hence, we add the state s_{21} and two transitions $s_1 \xrightarrow{c_0,a,a_0;a_1} s_{21}$ and $s_3 \xrightarrow{c_5,e,a_5;a_1} s_{21}$ over constrained by the post-condition of the $\epsilon-$transition a_1.

At this level, we can remove the $\epsilon-$transitions with having the behavior in the UCEA in Figure 62 (c). Due to the fact that pre-conditions have to be propagated forward and post-conditions have to be propagate backward, it is necessary to verify that the pre-condition

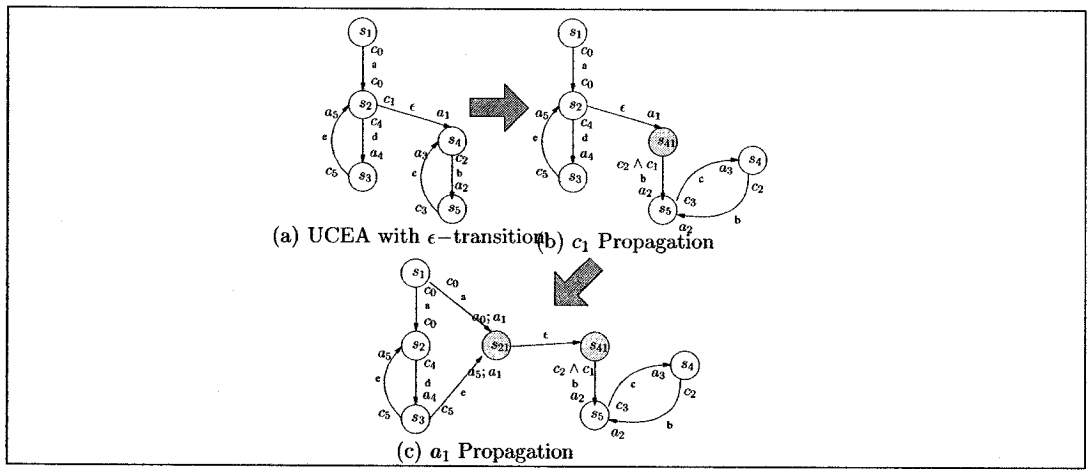


Figure 62: Constraints Propagation

and the post-condition of any $\epsilon-$transition are independent before processing to this step ($\epsilon-$transition removal).

However, the algorithm of *begin* and *end* transitions Removal cannot be applied in all cases. There is some restrictions related to the composition itself. In fact, when the outgoing transition of *begin* is an initial state of the base use case. Since the assignment specified in the composition expression has to be verified before starting the behavior of the base use case, we have no way to propagate backward the input assignment of the *begin* transition. In this case, either the modeler is advised in order to keep the *begin* transition and rename it with different labeling, or s/he has to change the composition expression with no input assignment. We believe that choosing one of the solutions is up to the needs of the modeler and the application. For instance, when the referred use case needs the initialization of some input variables, the first solution is the only applicable one if the modeler needs to remove the $\epsilon-$transitions.

We draw the attention that when removing *begin* or *end* transitions, we are duplicating some states and transitions (Figure 63 line(13,23)) . However, we are not duplicating any

(1) Input: a UCEA $A = (S, s^0, S^f, L \cup \{begin, end\}, V, I, E)$ with *begin* and *end* transitions
(2) output: a UCEA $A' = (S', s^0, S'^f, L, V, I, E')$
(3) Let $S' = S$; $S'^f = S^f$; $E' = E$
(4) For each $t = (s_1, c_1, a, c_2, s_2) \in E$ such that $a \in begin, end$ do
(5) If $a = begin$ then
(6) If $IngoingToState\ s_2 \neq \{t\}$ then
(7) /* there exists a transition different than t that is an ingoing transition to s_2 */
(8) $s'_2 = Duplicate\ s_2$
(9) If $s_2 \in S^f$ then
(10) $S'^f = S'^f \cup s'_2$
(11) done
(12) $S' = S' \cup s'_2$
(13) $Reconstitute\_IngoingTransitions\ of\ s_2\ in\ s'_2$
(14) $E' = E \cup AddedTransitions\ -\ \{t\}$
(15) /* Added transitions for the reconstitution of cycles in s'_2 */
(16) If $IngoingToState\ s_1 \neq \emptyset$ then
(17) /* there exists a transition different than t that is an ingoing transition to s_2 */
(18) $s'_1 = Duplicate\ s_1$
(19) If $s_1 \in S^f$ then
(20) $S'^f = S'^f \cup s'_1$
(21) done
(22) $S' = S' \cup s'_1$
(23) $Reconstitute\_IngoingTransitions\ of\ s_1\ in\ s'_1$
(24) $E' = E \cup AddedTransitions$
(25) /* Added transitions for the reconstitution of transitions in s'_1 */
(26) $MoveBackward\ a_1\ in\ IngoingTransitions\ s'_1$
(27) done
(28) $MoveBackward\ a_1\ in\ IngoingTransitions\ s'_1$
(29) done
(30) If $a = end$ then
(31) $MoveForward\ c_1\ in\ OutgoingTransitions\ s_2$
(32) done
(33) $E' = E' \cup t = (s'_1, true, \epsilon, true, s_2)$
(34) done

Figure 63: *begin* and *end* Removal Algorithm in the Case of UCEA

transition labeled with *begin* and *end*. This is due to the composition itself. The outgoing state of *end* and the ingoing state of *begin* are never the same. The first is labeled with the state of the base builder where we start the insertion of the referred behavior while the second is labeled with the state of base builder where we finish the insertion of the referred behavior.

8.5 Application to Multiple Extension Points in the Case of UCEAs

We presented our approach in the case of a single extension point. However, we can extend it to consider the case of specifying multiple extension points, as explained in the case of UCAs. Here again, a pre-processing of the UCEAs has to be made before generating the builders. This pre-processing generates clones of the UCEAs that will be used to generate the different builders. In this case, a clone of a UCEA is defined formally as follows:

Definition 16. *Let* $X = (S, s^0, S^f, L, V, I, E)$ *an UCEA. The clone of* X *with respect to a renaming function rename is an UCEA* $X' = (S, s^0, S^f, L', V', I', E')$ *such that :*

- $L \cap L' = \emptyset$
- $\forall e = (s_1, c_1, l, s_2, c_2) \in E,\ \exists e' = (s_1, c_1, rename(l), c_2, s_2) \in E'$
- $\forall v \in V, \exists v' \in V' / v' = rename(v)$
- $\forall i \in I, \exists i' \in I' / i' = rename(i)$

It denotes that a clone of a UCEA is a UCEA that has exactly the same behavior with a renaming of the labels and the variables, which differ from the definition of clone in the case

of UCA. Variables have been renamed because when inserting different clones of the same UCEA in different extension points, the clones are considered as independent. Renaming the variables preserve this independence, while in the opposite, values of a variables given in a clone can be used when executing a second clone.

8.6 Executability of the Resulting UCEA from Composition

When inserting a UCEA in an extension point, there is no guarantee that this UCEA will be executable in that point. By executable, we mean that all the transitions of the UCEA are executable. This comes from the fact that the conditions of the UCEA that appear in the outgoing transitions of the use case initial state may never be true in the extension point. Mainly, this non execution results from the possible values of the different variables specified in the base use case when reaching the extension point.

To illustrate this, let's consider the UCEAs of figure 64. The variable x is an integer. We note that both UCEAs are executable. By executable, we mean that it exists values of x where all the transitions of the UCEA are executable. In the first UCEA, $x = 0$ as input will lead to the execution of the trace $F.G$, while $x = 3$ as input will lead to the execution of the trace $F.H$. Hence, we can conclude that the UCEA X is executable. $x = 4$ as input to the UCEA Y will lead to the execution of the trace $A.B$. Consequently, Y is considered as executable. However, resulting UCEA from composition is not. This is due to the fact that the condition $x > 3$ will never be satisfied in state z_2. When reaching z_2, x will be equal to 0, 1 or 2. Consequently, the transitions $(z_2, x > 3, A, true, z_4)$ and $(z_4, true, B, true, z_5)$ will never be executed.

In our approach of composing, we believe that the resulting UCEA needs verification of

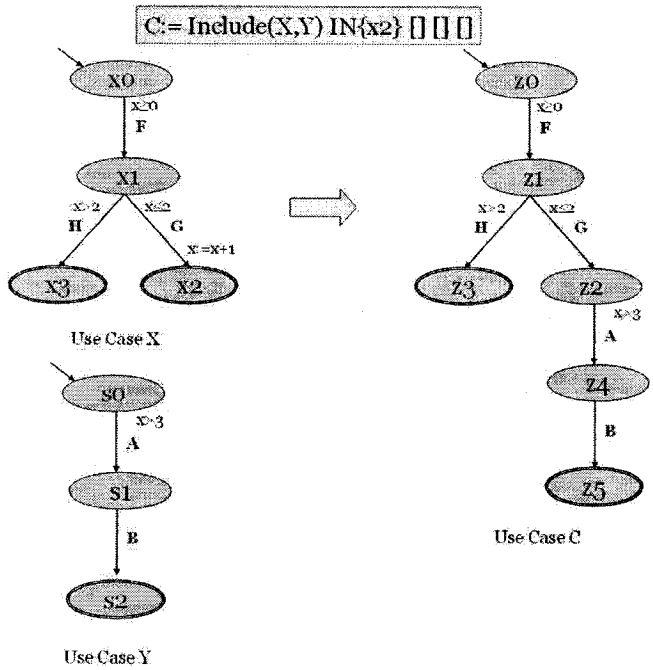


Figure 64: Example of a Non-executability of a Resulting UCEA from Composition

its executability. We propose to do it using the model checker. Such verification will help the analyst in building a more concise specification of the system.

8.7 Summary

In this chapter, we extended our formal and incremental approach for composing different user requirements expressed as extended finite state automaton. The introduction of variables in the description of use cases in the form of UCEA provides a concise, expressive, and scalable use case model. It also allows the generation of a more realistic specification of the overall system specification.

The variables can be defined either globally to the specification, as specification variables, or locally to the use cases, as UCEA variables. This distinction between variables contributes in making easier the task of writing the composition expressions, and allows the

reduction of the variables used in the description of the system. By adding another degree of expressiveness to our approach, we had to redefine the rules of generating builders. In addition, the composition expression had to manipulate the value passing between the local variables of the referred and base use cases, respectively.

Chapter 9

Use case Modeling And Composition Tool

9.1 Introduction

In order to validate our approach of composition, a tool has been developed by Siamak Kolahi, a master student at Concordia University, that implements the composition approach we presented so far. It is a tool for modeling, composing, and verifying use cases. Formal behaviors are designed through a visual graphical design layer, composed within the composition engine, verified on the model checking module and possible violations are parsed and visualized as a use case trace.

UMACT allows the description of use cases as UCAs or UCEAs. The composition engine is based on the label matching mechanism we presented. Definition of new behaviors is done through the specification of composition expressions that are evaluated. The resulting use case would be added to the existing set of use cases, serving as a new behavior in the system

specification. Evaluation of each list of composition expressions forms an increment of the system specification. Newly generated use cases are subjected to verification within a model checker that is linked to our tool.

In the first part of this chapter, we will give an overview of the tool, that is completely described in [55]. In the second part, we will use the tool in the generation of a state-based specification of the e-purchasing system using the UCA model first, and then using the UCEA model.

9.2 UMACT Tool Overview

The UMACT tool is implemented in Java and it consists of three layers. Each of the three steps of modeling, composition and verification is implemented in a certain layer, with distinct interfaces for future extensions of the platform and features for applying the theory of approach on real-life specification problems. Figure 65 shows the different layers of the tool and their components.

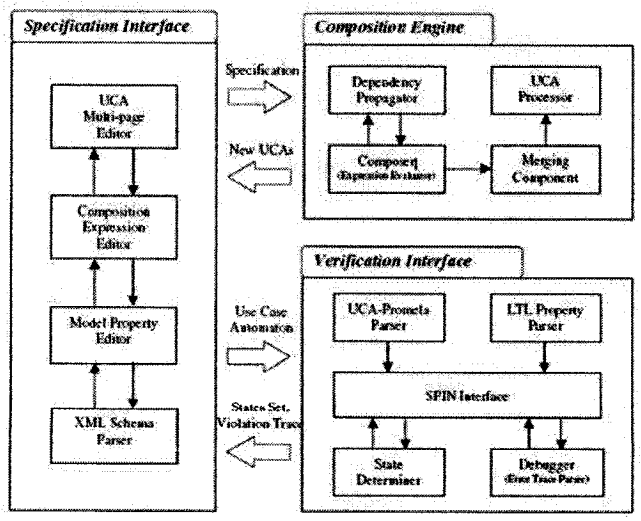


Figure 65: Tool Component Architecture [55]

9.2.1 Specification Interface

In the first layer, the specification of the requirements of the system in each increment is edited by the analyst. This includes adding new use cases to the set of existing ones, specifying the list of new composition expressions to compose new behaviors, editing the existing use cases and verifying recently created ones during the evolution of the system requirements specification. The tool interface is shown in Figure 66.

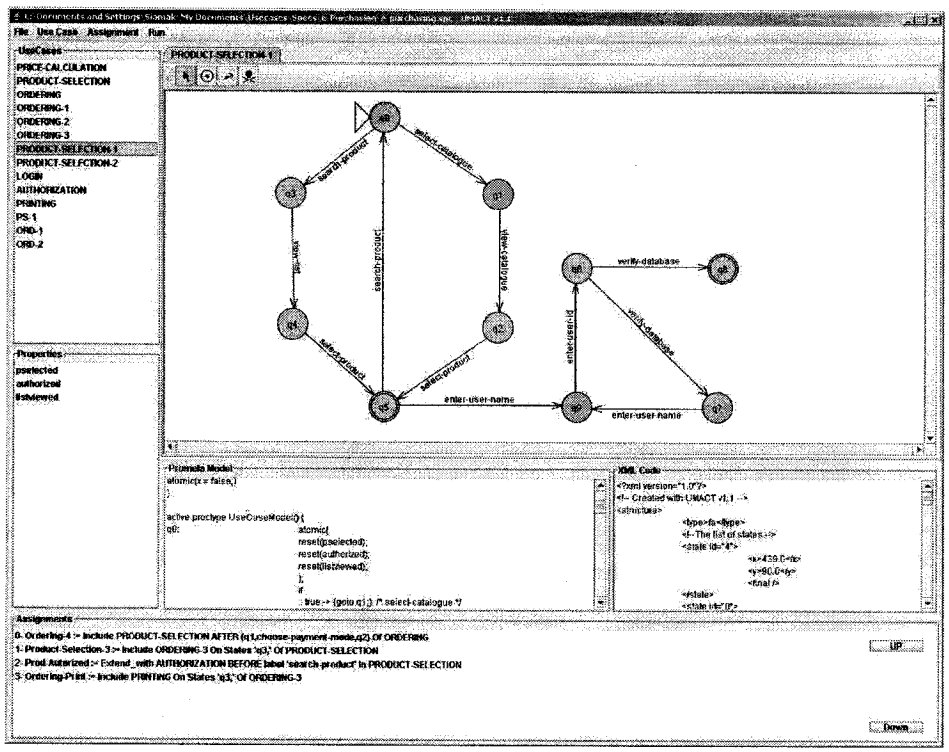


Figure 66: UMACT Tool Interface

The graphical view of the automata design component is based on JFLAP [8], a package of graphical tools which can be used as an aid in learning the basic concepts of Formal Languages and Automata Theory. JFLAP was customized for our model of use cases. An XML code is generated from graphical structure for the use in the composition and in the model checking phases.

The tool provides a multi-page editor of the specification layer, offering useful editing features such as creating automata from a set of words, minimizing created automata, and behavioral trace highlight for validation of each scenario of the partial requirements. An automata animator is also provided. It offers an interactive environment for animating the behavior of the automaton by stepping through certain sequences of actions. These features help the analyst to visualize more the behavior s/he specified in the different use cases.

A list of composition expressions can be defined interactively through the specification interface. A generic abstraction of the operators has been implemented which should be refined and overwritten for each composition semantics. This abstraction provides a common ground for deriving future user-defined operators with intended customized semantics, and gives the flexibility for updating the semantics of the existing ones. So far, we have defined six operators in the tool, named as *Include*, *Extend_with*, *Alternative*, *Graft*, *Interrupt*, and *Refine*. Moreover, the tool provides an XML schema in order to introduce new builders.

Finally, in order to facilitate the work of the anyalsyt, a coloring feature is implemented in the tool. When editing the use case, a color is assigned to it that will be used to color its states. When the use case is used in a composition expression, this color is maintained in the newly generated behavior. It a way to make the analyst retrieve the base behavior and the referred behavior in the new one. In addition, this feature can help in the validation task. It may help the analyst to retrieve the origin of a non desired behavior.

9.2.2 Composition Engine

The steps of the composition in each increment are performed within the composition engine of the tool. After specifying an increment for requirements specification, the composition engine is triggered and processes the increment by evaluating the list of composition expressions.

Each expression is evaluated within an instantiation of an expression evaluator component called *composer*. The semantics of the composition of each expression is implemented on these composers. This include the generation of the referred and base builders. The *merger* component provides the implementation of the merging algorithm based on the label matching mechanism. The result of the composition of builders is an intermediate automaton which needs further refinements to form the new partial behavior. The *UCA processor* component performs the post processes on the intermediate automaton: the determination of the final states which is done according to the semantics of the composition, the recovering of transition labels which were renamed during the cloning, and the *begin* and *end* transitions removal. The automaton product of the UCA processor component is the new model which is to be added to the existing set of use cases and serving as the new behavioral requirement for the specification.

9.2.3 Model Verification

Verification of behavioral models is achieved with the model checking layer of the UMACT tool. This interface is deployed for two distinct purpose: verification of model compliance over temporal properties and specifying the states of the model holding certain properties for the use as the extension points of the composition.

The UMACT has been interfaced with the SPIN model chercker [7] for verifying use cases. Temporal properties are specified as Linear Temporal Logic (LTL) formalism. As the first step, the use case is parsed to a proper Promela model for SPIN using the *UCA-Promela* parser component. Furthermore, the desired LTL property is specified through a *property manager* component and some predefined property templates. The SPIN is then triggered and the *debugger* component performs the analysis on the output. When the property is not verified, the counterexample given by SPIN is parsed back as a trace of the use case model, and is visualized on the respective use case in the specification layer, giving the possibility to the analyst to visually track the failure scenario.

Model verification is also used for determining states as extension points for the composition expression. Given a specified temporal property, the *state determiner* component of the tool determines the set of states holding that property using a surfing algorithm of checking the model over each state for the property. The *state determiner* communicates with the *verifier* component for each state, and determines if the state satisfies the intended temporal property, and builds a set of such states of the model. This set would form the extension points of the expression in the specification interface and would be evaluated in the composition engine for the builder generation process.

9.3 Traceability

While the two behaviors are composed and a whole new behavior is produced, tracking the generating behaviors would be a useful feature for further study of the requirements in the compound models. This feature would specifically be useful for tracking the changes in the basic requirements and their effect in the compound behavioral requirements.

Once a UCA is composed and added to the existing set of UCAs, it becomes a part of the specification as a new partial requirement of the system. However, this use case is not a basic one, it results from a composition operation. Accordingly, the set of UCAs in the specification in each increment can be categorized to two classes of use cases: user-defined use cases, we call *atomic* use cases, and the ones composed of others, we call *dependent* use cases. The structure of dependent use cases is in fact dependent on their precedences, which is not the case for atomic use cases.

Different increments in the specification create a hierarchy of composed use cases, which are dependent on each other. In fact, each use case in the specification may be composed of two use cases, each of them in turn may be composed from other use cases. In order to trace changes in the requirements, it is necessary to take into consideration such dependency. The tool implements this dependency in a *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 use case is inserted into the hierarchy, and updated forward and backward.

The hierarchical dependency chart is used in order to change dependent use cases when one of its depending use cases has been changed. This means that any change in the structure of a use case through the automata design interface can affect the behavior of all the forward depending use cases 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 the analyst can choose to propagate the change, or to destroy the dependency relationship of the changed use case and have it as an atomic one.

9.4 e-Purchasing System Specification in UCA Model

We apply our approach on generating a behavioral model for an e-Purchasing system. We first modify the specification in order to make it fit the model of use cases we presented in chapter 7. e-Purchasing systems cover a wide range of use cases representing purchasing activities. We still focus on use cases that emphasize activities representing the behavior of the system from the purchaser's side. Use cases related to other sections such as catalog maintenance and report managing are not considered.

Let's consider this informal specification:

e-Purchasing requires a number of activities to be performed. First the buyer has to select a product. She/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. After logging in and placing an order, the customer has to specify the information about the delivery, may print a quote, and eventually make the payment.

Figure 67 shows some UCAs that represent certain functionalities of the e-Purchasing system. Each UCA focuses on a single functionality. As it was the case in chapter 5, *Login* UCA describes the authentication of the client in order to be able to place an order. *Prod_Select* UCA shows the scenarios for choosing a product. *Delivery* UCA describes the scenario of entering information about the client address. *Order* UCA depicts the fact of placing an order by giving the quantity the client desires as well as the verification of the availability of this product in the inventory. *Printing* UCA expresses the printing of a quote to the client. Finally, *Price_Calculation* UCA determines the price of the order the client made.

It is to note that the sets of labels of the different use cases are disjoint. When it is not

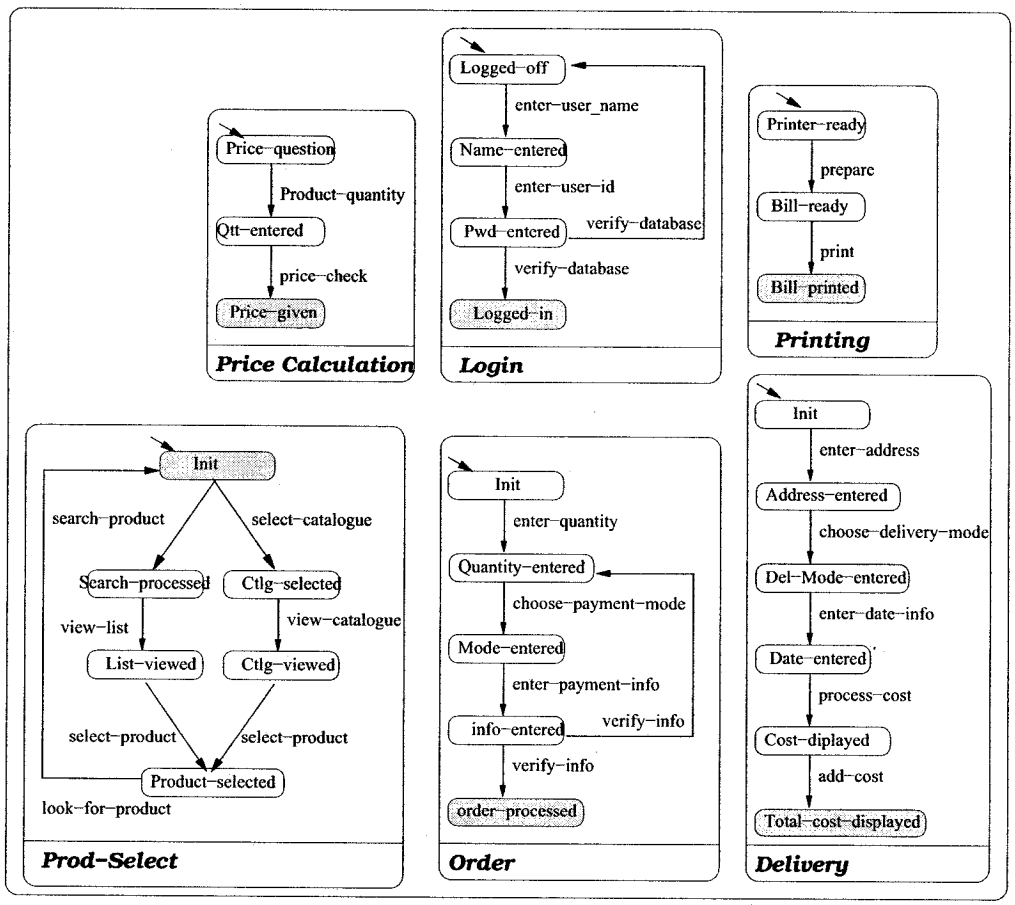


Figure 67: e-Purchasing System UCAs

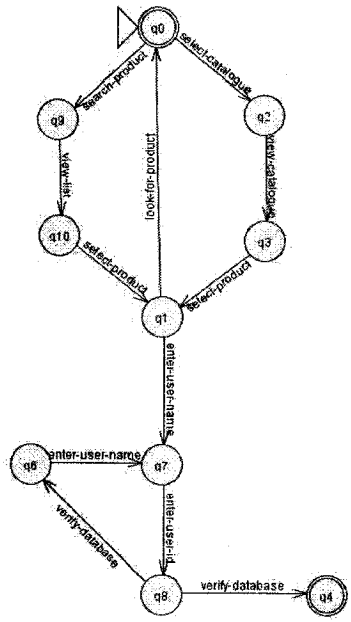


Figure 68: *Prod_Select_1* UCA

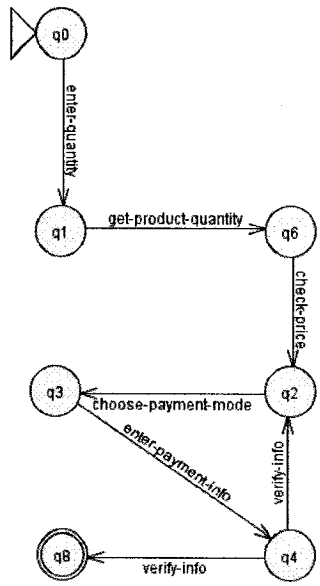


Figure 69: *Order_1* UCA

the case, the use cases are pre-processed in order to make the sets disjoint. After composing, a recovering of the original labeling is consequently needed. Let's construct incrementally a possible overall behavioral UCA of the described e-Purchasing activities. According to the given informal description, we will proceed to three increments. In the first, we create two new UCAs: *Prod_Select_1* and *Order_1* using the following expressions:

$$Prod\_Select\_1 := Alternative(Prod\_Selection, Login)\ IN\ \{Prod\_Selected\} \tag{63}$$

$$\begin{aligned} Order\_1 &:= Include(Order, Price\_Calculation)\ AFTER\ \{t\} \\ &where\ t = (Init, enter\_quantity, quantity\_entered) \end{aligned} \tag{64}$$

These two composition expressions form the first increment of the overall system synthesis. We draw the attention to the fact that they are independent. It means that they

can be evaluated in any order. Figure 68 shows the generated UCA using the composition expression 63. q_1 is in fact the state where the *login* UCA has been inserted with *Alternative* semantics. It expresses the fact that, after selecting a product, a user has the choice to proceed to the login to the system or to reselect another product. We notice that the final states of the obtained UCA are the union of the final state of the base UCA and the referred one.

Figure 69 shows the generated UCA using the composition expression 64. It includes after the first transition the use case *Price_Calculation*. The final states of the obtained UCA are those of the base UCA.

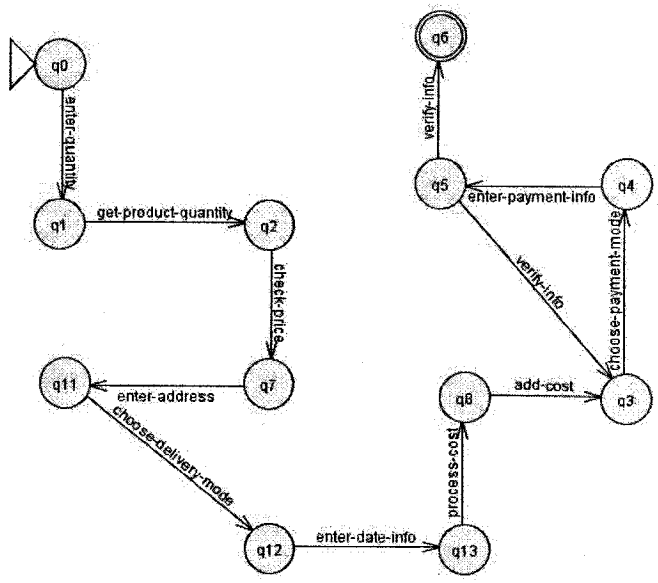


Figure 70: *Order_2* UCA

In the second increment, we generate the UCA that takes into account the customer's delivery information before making the payment. We use UCA *Order_1*. As illustrated in Figure 70, $(q6, check\_price, q7)$ is the extension point transition of *Order_1* UCA where we want to include Delivery UCA.

$$Order\_2 := Include(Order\_1, Delivery)AFTER\{(q6, check_price, q7)\} \quad (65)$$

Finally, we generate two UCAs during the third increment, *Order_3* and *Prod_Select_2*. The first, shown in Figure 71 allows a possible printing of the purchasing quote before making the payment and the second includes the behavior of *Order_3* after the login in *Prod_Select_1*.

$$Order\_3 := Extend\_with(Order\_2, Printing)\ IN\ \{q3\} \quad (66)$$

$$Prod\_Select\_2 := Include(Prod\_Select\_1, Order\_3)\ IN\ \{q4\} \quad (67)$$

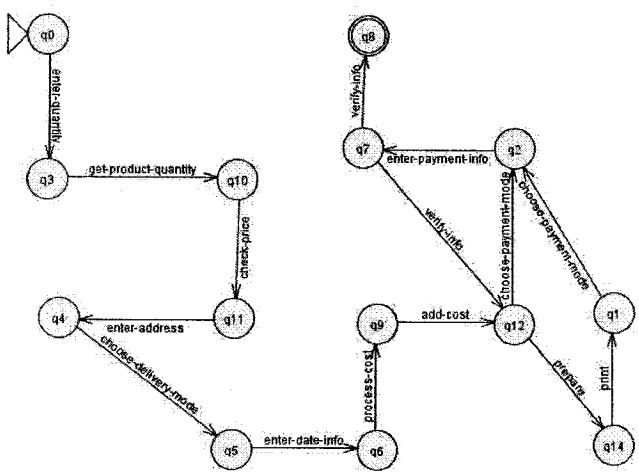


Figure 71: *Order_3* UCA

Prod_Select_2 models the steps to buy a product. The output of the UMACT tool is shown in Figure 72. We note that the *Prod_Select_2* UCA does not represent a complete model of the overall system behavior. More increments still have to be made to take into account other behaviors. We draw the attention that an increment could contain only a set

of expressions that are independent from the constructed UCA in that increment. In fact, in order to write the composition expression of the UCA *Order_3*, we need to have *Order_2* constructed because of the specification of the extension points.

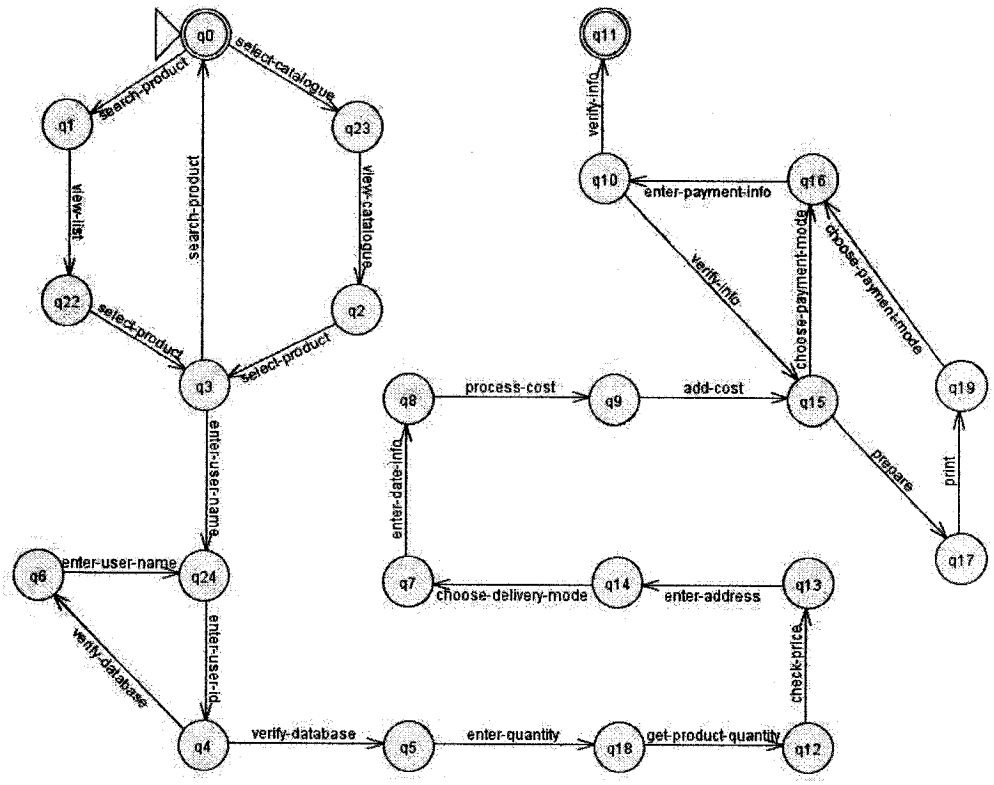
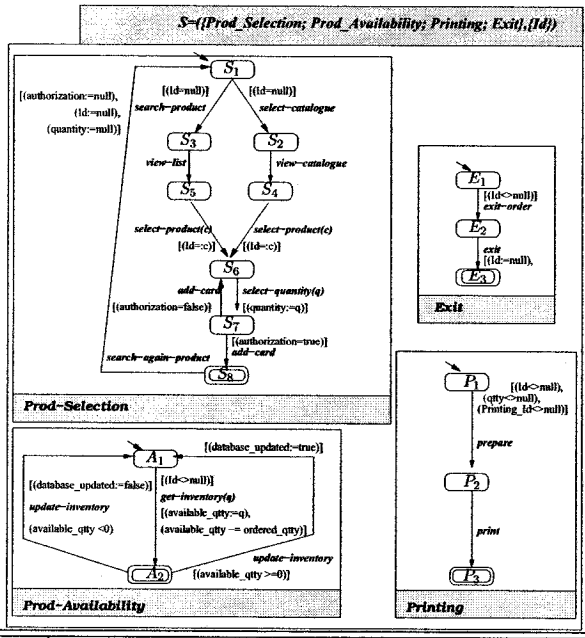


Figure 72: *Product_Select_2* UCA

9.5 e-Purchasing System Specification in the case of UCEAs

Let's now modify the specification of the e-Purchasing in order to make it more intuitive. In what follows, we present the same specification of the e-Purchasing using the extended model with variables (c.f. Figure 73).

We consider four use cases: *Prod_Selection*, *Prod_Availability*, *Printing*, and *Exit*. Since we are differentiating between specification variables and UCEA variables, we define



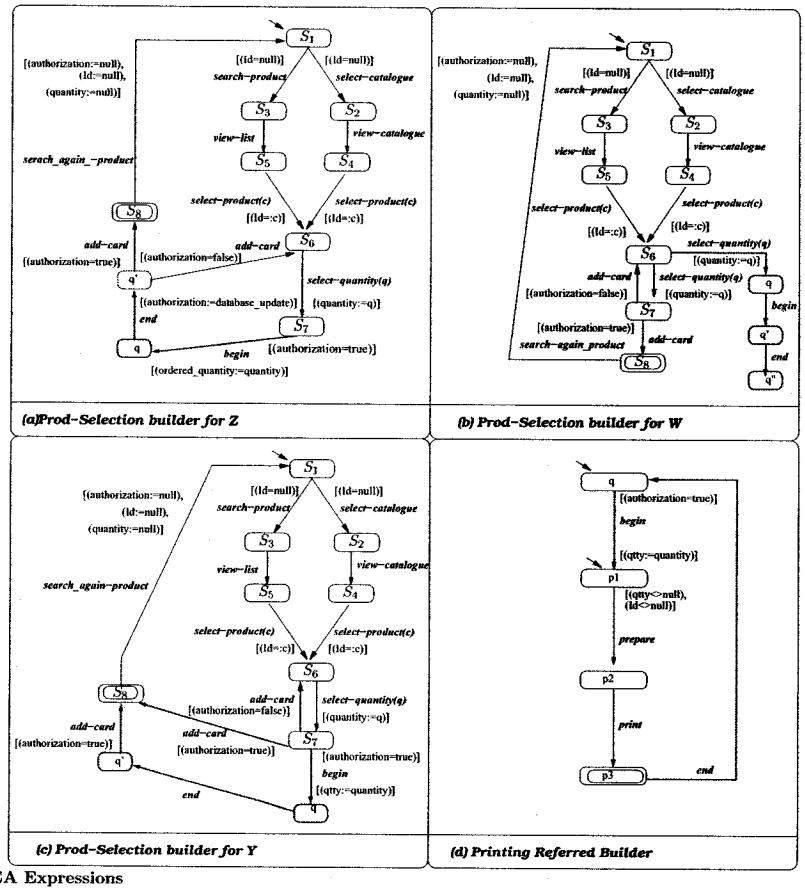
Use Case	Variables	Input Variables
Prod_Selection	$\{quantity,$ $authorization\}$	$\emptyset$
Prod_Availability	$\{available\_quantity,\}$ $database\_updated, ordered\_quantity\}$	$\{ordered\_quantity\}$
Printing	$\{qtty\}$	$\{qtty\}$
Exit	$\emptyset$	$\emptyset$

Figure 73: Some UCEAs of an e-Purchasing System

a specification variable *Id*. It defines the identifier of the product the customer has chosen. The first use case describes the activity of selecting a product from a list of a catalog. The variable *quantity* stores the quantity of the product the customer is ordering. *authorization* is a Boolean variable which keeps track whether the quantity asked by the customer is available in the inventory or not.

Prod_Availability UCEA checks for the availability of the product in the inventory. It has three variables: *available_quantity* returns the quantity that figures out in the database, *ordered_quantity* is a variable that indicates the quantity needed, and finally *database_updated* is a Boolean variable that indicates if the quantity in the database has

been updated by the new available quantity. *Exit* use case specifies the cancellation of the ordering. The *Printing* use case prints a quote for the customer. It has one input variable *qtty* that indicates the quantity of the product asked by the user.



UCEA Expressions

$Z :=$ $Include(Prod\_Selection, Prod\_Availability) IN\ (S_7)\ []$
$[(Prod\_Availability.ordered\_quantity := Prod\_Selection.quantity)]$
$[(Prod\_Availability.authorization := Prod\_Availability.database\_update)]$

$Y :=$ $Extend\_with(Prod\_Selection, Printing)\ BEFORE$
$(s_6, (authorization = true), add\_card, true, s_7)(Prod\_Selection.authorization = true)$
$[(Printing.qtty := Prod\_Selection.quantity)]\ []$

$W :=$ $Alternative(Prod\_Selection, Exit)\ AFTER\ (S_6, true, select\_quantity, (quantity := q), S_7)\ [][][]$

Figure 74: Illustration of Base and Referred Builders of the e-Purchasing in the Case of UCEAs

Figure 74(a) illustrates an example of a synthesized base builder for the use case *Prod_Selection* using the expression of Z given in the bottom of the figure 74. Figure 74(b) and (c) shows examples of base builders in the case of *Extend_with* and *Alternative*

operator, respectively. Figure 74(d) illustrates an example of a synthesized referred builder for the use case *Printing* using the expression of Y.

Figure 75 shows the newly generated use case Y after removing the $\epsilon-$transitions. It can be further used in the description of other UCEA expressions.

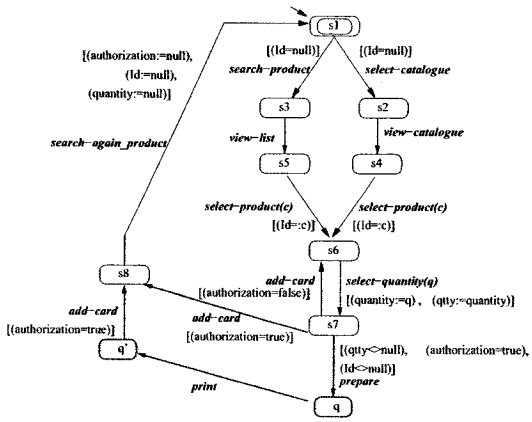


Figure 75: The Derived Y UCEA by composing *Prod_Selection* and *Printing* UCEAs

9.5.1 UCEA vs. Specification Variables

While specification variables are shared between UCEAs of the specification, UCEA variables play the role of local variables to the use case. Despite the dependency the specification variables create between use cases, they are actually needed in order to reduce the number of pass of the frequently-used variables throughout several UCEAs. They also permit the reduction of the total number of variables in the system specification.

As an example, let's consider the case of e-Purchasing system by the specification of UCEA variables (no specification variables is defined).

In the first use case, we had to define three variables. *prod_Id* defines the identifier of the product the customer has chosen. The variable *quantity* stores the quantity of the product the customer is ordering. Finally, *authorization* is a Boolean variable which keeps

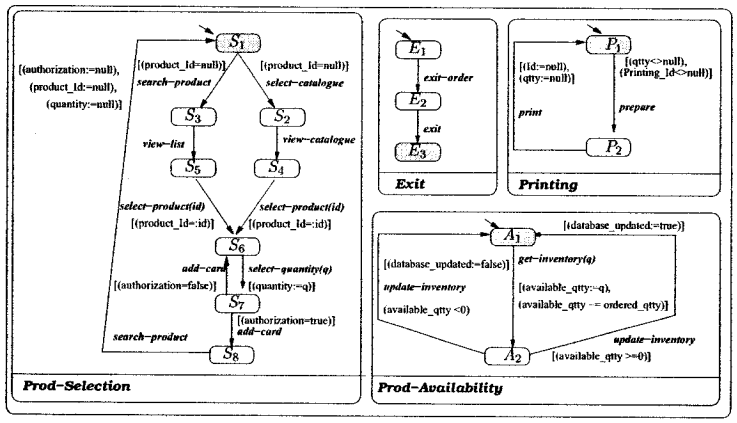


Figure 76: Use Cases of the e-Purchasing System in the Case of Local Variable Specification

Use Case	Variables	Input Variables
Prod_Selection	$\{prod\_Id, quantity, authorization\}$	$\emptyset$
Prod_Availability	$\{Id, available\_quantity, \}$ $database\_updated, ordered\_quantity\}$	$\{Id, ordered\_quantity\}$
Printing	$\{qtty, Printing\_Id\}$	$\{qtty, Printing\_Id\}$
Exit	$\emptyset$	$\emptyset$

Table 5: UCEA Variables of the e-Purchasing Use Cases

track whether the quantity asked by the customer is available in the inventory or not.

Prod_availability UCEA has four variables: *Id* represents the identifier of the product for which availability would be checked, *available_quantity* returns the quantity that figures out in the database, *ordered_quantity* is a variable that indicates the quantity needed, and finally *database_updated* is a Boolean variable that indicates if the quantity in the database has been updated by the new available quantity.

The *Printing* use case has two input variables, *qtty* and *Printing_Id*, that indicate the identifier and quantity of the product asked by the user, respectively. Hence, in total, we defined ten UCEA variables against six UCEA variables and one specification variable in the specification in Figure 76. In order to define the equivalent composition expression of

the UCEA Y, we have to write now:

$$Y := Extend\_with(Prod\_Selection, Printing)\ BEFORE$$

$$(s_6, (authorization = true), add\_card, true, s_7)$$

$$[(Printing\_Id := prod\_Id), (qtty := quantity)]\ ([])$$

This equation passes the values of the product identifier as well as the asked quantity of the product as *Input_Var_Assign*. However in the equation of Y in Figure 74 with specification variables we had to pass only the product quantity asked. Consequently, the specification variable *Id* has made the task easier for the modeler since she/he has to manage less variable passing and therefore reduced the complexity of writing the appropriate composition expressions.

9.6 Summary

To validate our approach, we have implemented the UMACT tool. It is a tool for editing and composing use cases based on the concepts of expressions, label matching, and builders. In addition, UMACT provides some features that facilitate the process of generating the system behavioral model. It allows to graphically edit UCAs and UCEAs, to track the depending UCA (UCEAs), and to reflect changes of a UCA (UCEA) in all of its depending UCAs (UCEAs)- an especially useful feature for the maintenance of the behavioral model. This traceability feature is basically allowed because of the incremental nature of the specification generation. In fact, the order in which the expressions of composition are evaluated tracks the dependency between the different use cases, an interesting feature in requirements gathering and analysis.

UMACT has also an interface with the model checker SPIN in order to validate and

verify the obtained use cases by composition against their correctness. We used this tool for the construction of a formal specification of an e-Purchasing system. We have experimented with the construction of the specification using UCA and UCEA models.

While with UCEAs the use cases were more realistic, variables may complicated the process of generating a system specification. First, decision about the type of each variable has to be made. Second, composition expressions have to specify the assignments between local variables of the UCEAs. Despite this difficulty, we believe that the obtained specification is closer to design and can be used for more reliable verification of the system requirements.

Chapter 10

Conclusion

As described in section 1.3, this thesis addresses the problem of defining a formal use case model as well as an automated composition approach that helps the generation of a formal system specification. In this chapter, we summarize the main contributions of the thesis, discuss their applicability, and give a list of directions for future research.

10.1 Contributions

We have presented in this thesis many contributions related to the formal composition of use cases.

10.1.1 Implicit Composition of Use Cases

In a first contribution, we have tackled the problem of composing use cases using the state characterization. Use cases are represented as variant of extended finite state machines. We differentiate between the location variables, used to characterize the states of the use case, and control variables, which describes the control part of the use cases. The use

cases are overlapping. The overlapping parts are merged in the composite automaton. Contrarily to what exists in the literature, our use case model is expressive. It allows the description of explicit loops. These loops are protected during the composition in order to avoid introducing on their bodies implied scenarios that may threaten their overall behavior.

In implicit composition, the modeler needs to have a certain expertise and a deep understanding of the overall system in order to specify the adequate characterization of the states in the different use cases. This characterization is in fact crucial in obtaining the right system behavior. While composing overlapping use cases may help the generation of the system automaton in certain applications where traces of the overall system are described, this approach of modeling is against the tendency of separating the different concerns of the system early in the system lifecycle.

10.1.2 Explicit Composition of Use Cases with Interactions

In a second contribution, we have considered the explicit composition of use cases by means of interactions. An interaction is an invocation between two use cases.The interactions that a use case makes with the other use cases in the specification are specified within the use case itself. When composing, a state based model interpreting the specified interactions within a use case is generated. The obtained state-based models are then merged together. In order to avoid implied scenarios, a graph of interactions is generated from the specified use cases in order to discover interferences. Interferences are non specified interactions between use cases. When interferences are discovered, control variables are added to the obtained system state model in order to remove them.

The approach presented has the advantage of helping the separation of concerns. It also has the advantage of being fully automated and does not introduce any implied scenario

in the generated overall system state based model. However, depending on the number of interferences, it may add a lot of additional variables, which contributes making the validation and the verification of the behavior harder. In addition, the fact that interactions are described within the use cases themselves constrains the modeling of the use cases.

10.1.3 Explicit Composition of Use Cases using Composition Expression

In a third contribution, we have proposed a novel approach of use case composition based on imperative expressions. These expressions are in fact representing the semantics of the composition operation the analyst wants to perform. Our approach of composition consists of three main steps. First, the analyst provides a set of use cases where each one defines a partial system behavior. Then, in an incremental way, the modeler can define new behaviors using composition expressions. These expressions are evaluated to derive new behaviors. Each expression specifies the two behaviors to merge, the composition operator, as well as the extension points. If the model is enriched with variables, conditions and assignments between variables of different use cases may be defined.

We have presented a formalization of the approach in the case of use cases modeled as a variant of finite state machines and variant of extended finite state machines. A tool implementing the approach has been developed. It offers a support that helps the designer modeling, composing, validating, and verifying use cases expressed as state based models.

The approach has the advantage of being fully automated and incremental -two important features in the requirements analysis phase, where the user-needs are prone to change. The use of composition expressions to define new behaviors makes the task of the modeler easier since he can experiment with many compositions until reaching the desired behavior. By construction, the composition does not introduce any implied scenarios, without over

constraining the model.

Introducing variables to the model of use cases has added a new expressiveness level. However, it also made the composition expressions more difficult to write. When experimenting with the two models (finite state machines and extended finite state machines) with the e-Purchasing specification, we had to make decisions about the scope of each variable, which needs a deep understanding of the requirements. Local variables make the description of the use cases easier since they can be described independently from each other, however they complicate the composition expressions. Global variables have to be managed properly in different use cases since they are shared, however they make easier the definition of the composition expressions. We believe that a trade off has to be made in order to keep the composition expressions as intuitive as possible.

10.2 Discussion on Future Work

10.2.1 Application of the Approach in the Case of Statecharts

As presented in this thesis, we have applied our method in the case of finite state machines and extended finite state machines. State Machine modeling is concerned with modeling the system which it describes as a collection of discrete sates. The model transitions from one state to another system state when stimuli are received. However, such a model represents some limitations such as the complexity of the state diagram and the lack of support of concurrency. In fact, The complexity of the model increases dramatically as the number of possible states increases. The state machine model then becomes unreadable and hard to use. Traditional state machine modeling is based on sequential transitions from one state to the next. Concurrent systems cannot be modeled in this manner as various aspects of

the system may be in different states. Statecharts overcome such limitations.

We plan to extend our approach in order to handle statecharts as use case formal model. In this way, we allow the description and the verification of the concurrent systems. We believe that the composition will still be based on imperative expressions. However, the label matching as well as builders may be revised because systems are no more considered as sequential.

10.2.2 Use Case Decomposition

Usually, a system is composed of a set of objects and use cases have to show the interactions among these objects in order to exhibit the expected system behavior. When use cases are described in a distributed way, the modeler needs to generate an FSM per object.

The FSMs of objects are assumed to be communicating FSMs. Each FSM object is autonomous and can communicate with other FSMs by means of message exchange. When an FSM object sends a message to another FSM, the latter is assumed to be ready to receive this message. The behavior of the overall system is represented by the parallel composition of the communicating FSMs. The verification will be performed regarding the interactions of the different communicating FSMs. It will allow the detection of common bugs such as the unspecified reception, service denial, and deadlocks that may have uncontrolled consequences on the behavior of the distributed system. When no such bugs are detected, the design of the system is validated with respect to the set of use cases originally specified by the modeler (and hence the overall system behavior described by the use case obtained after applying several composition increments).

As a future direction of our work, we propose to distribute our use cases according to an architecture. After composing, the generated use case representing the overall system

behavior has to be distributed according to the different objects. We propose to verify these communicating FSMs according to the original non distributed FSMs. Problems related to the distribution, such as the well known problems of deadlocks and unspecified receptions, have to be solved in order to obtain an equivalent behavior.

Having an FSM per object gives us the opportunity to link our tool to other modeling tools such as SDL (Specification and Description Language). SDL has already several tools that are linked to it. And hence the generated specification can be tested and verified within them. Moreover, code can be generated.

10.2.3 Approach Application

As a future work, we propose to apply our approach to several areas:

- **Application in Aspect Oriented Software Development**

  In addition to use cases approaches, aspect-oriented software development (AOSD) may be related to our work. Most of the research done in the area of AOSD has concentrated on developing methodologies that help the separation of aspects during the design and implementation phase. However, the separation of crosscutting concerns has to be supported across the development lifecycle at different levels of abstraction. Baniassad and Clarke [16] proposed the Theme approach that supports aspect-oriented development at the level of requirement and design. The direct mapping they are defining between the level of requirement and the level of design allows the maintenance of the traceability. In addition, the approach is based on the concept of a Theme which represents a collection of structures and behaviors that represent one feature. The notion of Themes is loosely similar to the notion of functionalities

identified by use cases.

Rashid et al. [84, 83] presented an Aspect-Oriented Requirement Engineering (AORE) model that supports the definition and composition of aspects through XML schemes. Similar to our approach, they are composing aspectual and nonaspectual requirements in order to constrain the behavior of the latter, using composition rules. The requirements are defined in a different abstraction level of our use case model. They are using an XML while we are using a formal model for use case which making the resulting use case easily validated using any formal validation approach.

As presented in the related work chapter, Arajo et al. [14, 107] presented an approach that focused on representing aspects during use case modeling. Two different models are used to describe aspects and nonaspectual scenarios: Interaction Pattern Specification and UML sequence diagrams. The composition is directed by merging directives and is done in the state-based model using the algorithms presented in [110]. Since we have extended the model of use cases with variables, we believe that our composition mechanism can offer a definition of the semantics of the weaving done in aspects.

As part of our future work, we are planning to define the semantics of AspectJ [53] with the operators we have defined since we have extended the model with data. A mapping between the advice weaving and the operators we are proposing when extending the model with data has to be developed. The validation of the weaving will be achieved through the validation of the automaton obtained by composition.

- **Application in Service Composition**

With the growth of Internet, web services have gained a lot of popularity. It is usual

that the implementation of a service involves the composition of several other services. The validation and the verification of the composed services are mandatory to understand their exhibited behavior. Models that represent web services and methodologies that compose them are needed. In addition, tools that support the verification of properties to confirm expected results from the viewpoints of the designer are required. The way of composing web services to generate composite services can be used in order to generate test cases for the verification of web services composition. Generating a state based model of the composite web services helps the testing and the verification of these web services. Patterns of composition in the state based model can be developed. They should represent the possible interactions between web services. We believe that introducing concurrency in the model will help the modeling of web service interactions. In addition, using variables in the modeling of web services can model some quality of service metrics.

- **Application in Program verification**

  Every program implicitly asserts a theorem to the effect that if certain input conditions are met then the program will do what its specifications says it will. The ability to prove mathematically that a program correctly implements its specification is increasingly important in ensuring that high integrity computer-based systems for security and safety-critical applications perform correctly.

  One of the methods used to formally verify software is to map the implementation level description of the software artifact mechanically to the description language of an existing verification tool [43]. The application is rewritten to match the input of a given verification tool. Java Pathfinder tool [6], the Bandera toolset [41], and the

FeaVer toolset [44] have been developed in order to verify programs using the SPIN model checker by generating from a program a SPIN input.

In a similar way, we aim at verifying programs using the composition approach presented so far. Since our tool is already linked to SPIN, we plan to extend the model of EFSM in order to be able to define all the entities that a program specifies. With the existing model, we have defined three type of variables: integer, boolean, and enumerations. Using these types, we can verify the call of certain functions. However, we are not able to handle more complex paradigms of object oriented programming, such as objects. Extension of the model in order to handle such paradigms is considered as part of our future work.

Appendix A

Equivalence of "BEFORE a state" and "AFTER a state" as Extension Point

We want to show that in our case, "AFTER a state" and "BEFORE a state" lead to trace equivalent automata. To do so, we have to prove that the base builders generated in both cases (c.f. Figure77) are trace equivalent. Let's consider the case of Include operator.

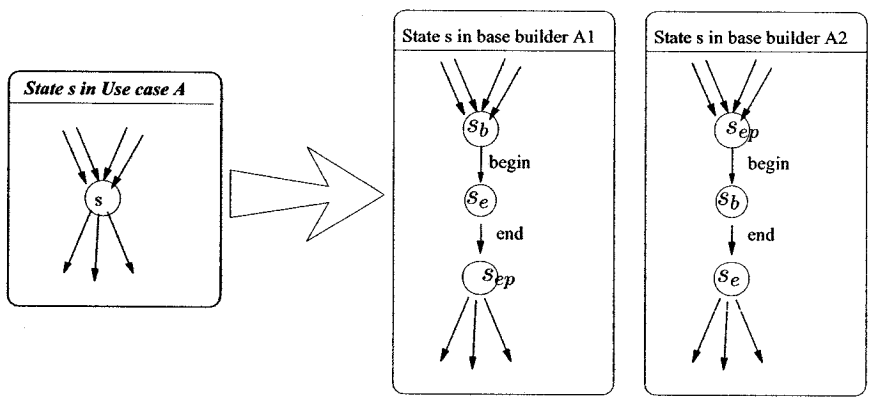


Figure 77: Examples of Builders for "BEFORE s" and "AFTER s"

Definition 17. *Let* $A = (Q, s^0, S^f, L, T)$ *the base UCA for the two automata builders generated for state* s_{ep}. $A_1 = (Q_1, s_1^0, S_1^f, L_1, T_1)$ *is the builder automaton generated AFTER the state* s_{ep}, *whereas:*

- $Q_1 = Q \cup \{s_b, s_e\}$
- $s_2^0 = s^0$
- $S_1^f \subset S^f \cup \{s_e\}$
- $L_1 = L \cup \{begin, end\}$
- $T_1 : Q_1 \times L_1 \longrightarrow Q_1$ *such that:*

$$(s_1, l, s_2) \in T_1 : (s_2 \neq s_{ep}) \wedge (s_1, l, s_2) \in T_1) \tag{68}$$

$$\vee\, (s_1 = s_{ep} \wedge l = begin \wedge s_2 = s_b)) \tag{69}$$

$$\vee\, (s_1 = s_b \wedge l = end \wedge s_2 = s_b)) \tag{70}$$

$$\vee\, (s_1 = s_e \wedge (s_e, l, s_2) \in T_1) \tag{71}$$

$$\tag{72}$$

Definition 18. *Let* $A = (Q, s^0, S^f, L, T)$ *the base UCA for the two automata builders generated for state* s_{ep}. $A_2 = (Q_2, s_2^0, S_2^f, L_2, T_2)$ *is the builder automaton generated AFTER the state* s_{ep}, *whereas:*

- $Q_2 = Q \cup \{s_b, s_e\}$
- $s_2^0 = s^0$
- $S_2^f \subset S^f \cup \{s_e\}$
- $L_2 = L \cup \{begin, end\}$

- $T_2 : Q_2 \times L_2 \longrightarrow Q_2$ *such that:*

$$(s_2, l, s_2) \in T_2 : (s_2 \neq s_{ep}) \wedge (s_1, l, s_2) \in T_2 \tag{73}$$

$$\vee (s_1 = s_b \wedge l = begin \wedge s_2 = s_e) \tag{74}$$

$$\vee (s_1 = s_e \wedge l = end \wedge s_2 = s_{ep}) \tag{75}$$

$$\vee (s_2 = s_{ep} \wedge (s_{ep}, l, s_2) \in T_2) \tag{76}$$

You can see these two automata in Figure 77, where A_1 is the builder automaton after state s and A_2 is the same automaton before the state s.

For the language of A, $L(A)$, we define two sets of $pref(L)$ and $post(L)$ to be the set of prefixes and postfixes of the $L(A)$, respectively. Based on these two sets, we define two other sets of $pref(L.s)$ and $post(L.s)$ as the set of prefixes and postfixes of the words passing by the state s. It is clear that for every word $w \in L(A)$ passing by a state s, w can be written $:w = u.v$, $u \in pref(L.s)$ and $v \in post(L.s)$.

Definition 19. *Let $w = l_0, l_1, l_2, ...$ be a word, and L a set of alphabet. The projection of w onto L, which we denote as $w_{|L}$ is the result of eliminating from the word w all the elements l_i in L. We call $w_{|L}$ as the free word from the alphabet L.*

Definition 20. *Let A_1 and A_2 two automata such that $L_1 = L_2 \cup \{begin, end\}$, and $e_1 = q_1, ..., q_n$ such that $q_i = (s_i, l, s_{i+1})$. Let $L_{be} = \{begin, end\}$ be an alphabet. Therefore, $e_2 = q'_1.q'_2....q'_{n-k} \in Ex(A_2)$ is called Execution projection equivalent of e_1, denoted as $e_2 \equiv e_1$ if and only if $e_2 = e_{1|L_{be}}$.*

Literally, e_2 is the same execution of automaton as of e_1, removing begin and end labeled transitions from its relevant word.

Definition 21. *For two automata A_1 and A_2, $L(A_1)$ is the Language Projection Equivalent to $L(A_2)$ if for every $w_1 \in L(A_1)$, there exists $w_2 \in L(A_2)$ such that $w_1 \equiv w_2$. We denote it as $L_1 \equiv L_2$.*

Lemma 1. *Let A be the base automaton used to generate the builder, and A_1 the builder automaton generated after the state s_{ep}, according to Definition 18. The two languages of $L(A)$ and $L(A_1)$ are language projection equivalent.*

Proof. First we want to show that for every $w_1 \in L(A_1)$, there exists a $w \in L(A)$ such that $w \equiv w_1$.

Suppose $w_1 = u.v$ such that $u \in pref(L_1.s_{ep})$ and $v \in post(L_1.s_{ep})$ as s_{ep} the state of the composition. We can write w_1 as $w_1 = u_1.(u_2.begin.end.v_2)^i.v_1$ with $(u_2.begin.end.v_2)$ as the loop passing by state s_{ep}, u_1, u_2, v_1, and v_2 are free of *begin* and *end*, and i the number of iteration of this loop. Therefore we have $u_1.u_2 \in pref(L_1)$ and $v_2.v_1 \in post(L_1)$ which does not contain transitions labeled as *begin* and *end*. We will show that it exists a $w \in L(A)$such that $w \equiv w_1$. Let e_1 the execution of w_1.

$e_1 = q_0.q_1...q_n.(q_{n+1}...q_m.q_b.q_e.q_{m+1}....q_l)^i.q_{l+1}...q_p$ where:

$q_b = (s_{ep}, begin, s_b)$, $q_e = (s_b, end, s_e)$, and $q_i = (s_i, l_i, s_{i+1}) \in T_1, 0 \leq i \leq p$

We show $e_{1|L}$, projection of e_1 onto the alphabet $L = \{begin, end\}$ as follow.

Since u_1, u_2, v_1, and v_2 are free of alphabet Laccording to Definition 18 ,

$e_{1|L} = q_0.q_1...q_n.(q_{n+1}...q_m.q'_{m+1}....q_l)^i.q_{l+1}...q_p$ where $q'_{m+1} = (s_{ep}, l_m, s_{m+1}) \in T$ and $q_i = (s_i, l_i, s_{i+1}) \in T, 0 \leq i \neq (m+1) \leq p$ Therefore, we have

$w = l_0, l_1, ..., l_n, (l_{n+1}, ..., l_m, l_{m+1}, ..., l_l), l_{l+1}, ..., l_p \in L(A)$ such that :

$w \equiv w_1$

Now we want to reversely show that for every $w \in L(A)$, there exists a $w_1 \in L(A_1)$ such that $w \equiv w_1$, or in other word, for every $w \in L(A)$ there exist $u \in pref(L)$, $v \in post(L)$, $q_b = (s_{ep}, begin, s_b) \in T_1$, and $q_e = (s_b, begin, s_e) \in T_1$ such that $w = u.v$, $w_1 = u.begin.end.v \in L(A_1)$, and $w_1 \equiv w$.

Suppose $w = q_1.q_2...q_n$.

For every $q_i = (s_i, l_i, s_{i+1} \in T)$, there is two cases. If $s_i \neq s_{ep}$, we have $u = w \in pref(L)$ and $v = \emptyset \in post(L)$ in which obviously $w_1 = u.v \in L(A_1)$. Otherwise we should have had $q_{i+1} = (s_{ep}, l', s_{i+1}) \in T$ and therefore since $q_b = (s_b, end, s_e) \in T_2$, we can have $u = l_1 l_2 ... l_{ep} \in pref(L)$, and $v = l_i.l_{i+1}....l_n \in post(L)$ which gives us $w_1 = u.begin.end.v \in L(A_1)$. □

Lemma 2. *Let A be the base automaton used to generate the builder, and A_2 the builder automaton generated before the state s_{ep}, according to Definition 17. The two languages of $L(A)$ and $L(A_2)$ are language projection equivalent.*

Proof. Same proof as the previous lemma. □

Theorem 1. *Let A_1 and A_2 the two base builders generated after and before the state s_{ep} of the base use case A according to Definition 18 and Definition 17, respectively. $L(A_1)$ and $L(A_2)$ are equivalent.*

Proof. First we show that $L(A_1) \subset L_(A_2)$.

Let $w_1 \in L(A_1)$ such that $w_1 = u_1.(u_2.begin.end.v_2)^i.v_1$ in which $u_1.u_2 \in pref(L_1.s_{ep})$ and $v_2.v_1 \in post(L_1.s_{ep})$ are free prefix and postfixes of the word w_1 on the alphabet $\{begin, end\}$, and the i is the number of iteration of the loop passing the state s_{ep}. Therefore,

from the Lemma 1, we know that there exist $w = u_1.(u_2.v_2)^i.u_2 \in L(a)$ such that $w \equiv w_1$. Having $w \in L(A)$, from Lemma 2, we conclude that there exist $w_2 = u_1.(u_2.begin.end.v_2)^i.v_1 \in L(A_2)$ such that $u_1.u_2 \in pref(L_2.s_{ep})$ and $v_2.v_1 \in post(L_2.s_ep)$, in which $w \equiv w_2$.

Accordingly, we can show that $L(A_2) \subset L_(A_1)$. Similarly, for every $w_2 \in L(A_2)$ according to Lemma 2, there exist such $w \in L(A)$ that $w \equiv w_2$. And from $w \in L(A)$ and according to Lemma 1, we conclude that there exists $w_1 \in L(A_1)$ such that $w_1 \equiv w$.

From $L(A_2) \subset L_(A_1)$ and $L(A_1) \subset L_(A_2)$, we have $L_1 = L_2$. □

We have proved the trace equivalent in the case of *Include* operator. Same approach is applied to prove the case of *Alternative* and *extend_with* operators.

Appendix B

Formal Definition of the Composition Operator

- **Case of $Include(A, B)$ and "after transition" extension point**

$$\frac{e \in ex(A) \setminus ex(A, ep)}{e \in ex(C)} \quad (77)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.ep.r \ where \ u \in Pref(A, ep), \ ep.r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.ep.e_b.r \in ex(C)} \quad (78)$$

- **Case of $Include(A, B)$ and "before transition" extension point**

$$\frac{e \in ex(A) \setminus ex(A, ep)}{e \in ex(C)} \quad (79)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.ep.r \text{ where } u \in Pref(A, ep),\ ep.r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.e_b.ep.r \in ex(C)} \quad (80)$$

- **Case of** $Extend\_with(A, B)$ **and "after transition" extension point**

$$\frac{e \in ex(A)}{e \in ex(C)} \quad (81)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.ep.r \text{ where } u \in Pref(A, ep),\ ep.r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.ep.e_b.r \in ex(C)} \quad (82)$$

- **Case of** $Extend\_with(A, B)$ **and "before transition" extension point**

$$\frac{e \in ex(A)}{e \in ex(C)} \quad (83)$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.ep.r \text{ where } u \in Pref(A, ep),\ ep.r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.e_b.ep.r \in ex(C)} \quad (84)$$

- **Case of** $Alternative(A, B)$ **and "after transition" extension point**

$$\frac{e \in ex(A)}{e \in ex(C)} \tag{85}$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.ep.r \ where \ u \in Pref(A, ep), \ ep.r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.ep.e_b \in ex(C)} \tag{86}$$

- **Case of** $Alternative(A, B)$ **and "before transition" extension point**

$$\frac{e \in ex(A)}{e \in ex(C)} \tag{87}$$

$$(e \in ex(A, ep));$$

$$(u, r \mid e = u.ep.r \ where \ u \in Pref(A, ep), \ ep.r \in Post(A, ep));$$

$$\frac{(e_b \in ex(B))}{u.e_b \in ex(C)} \tag{88}$$

- **Case of** $Graft(A, B)$ **and "BEFORE transition" extension points** ep_1 **and** ep_2

$$\frac{e \in ex(A)}{e \in ex(C)} \tag{89}$$

$$(e_1 \in ex(A, ep_1));$$

$$(u_1, r_1 \mid e_1 = u_1.ep_1.r_1 \ where \ u_1 \in Pref(A, ep_1), \ ep_1.r_1 \in Post(A, ep_1));$$

$$(e_2 \in ex(A, ep_2));$$

$$(u_2, r_2 \mid e_2 = u_2.ep_2.r_2 \ where \ u_2 \in Pref(A, ep_2), \ ep_2.r_2 \in Post(A, ep_2));$$

$$\frac{(e_b \in ex(B))}{u_1.e_b.ep_2.r_2 \in ex(C)} \tag{90}$$

- **Case of $Graft(A, B)$ and "AFTER transition" extension points ep_1 and ep_2**

$$\frac{e \in ex(A)}{e \in ex(C)} \quad (91)$$

$$(e_1 \in ex(A, ep_1));$$

$$(u_1, r_1 \mid e_1 = u_1.ep_1.r_1 \; where \; u_1 \in Pref(A, ep_1),\; ep_1.r_1 \in Post(A, ep_1));$$

$$(e_2 \in ex(A, ep_2));$$

$$(u_2, r_2 \mid e_2 = u_2.ep_2.r_2 \; where \; u_2 \in Pref(A, ep_2),\; ep_2.r_2 \in Post(A, ep_2));$$

$$\frac{(e_b \in ex(B))}{u_1.ep_1.e_b.r_2 \in ex(C)} \quad (92)$$

Appendix C

Synthesis Rules of Base Builders

Table 6: Builders Synthesis rules for UCA in the case of state extension point

$Z := Include(X, Y)\ IN\ \{s\}$ [a]

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_s(begin), f_s(end)\}, T)$

---

[a] s is a constant that represents the extension point (state)

$$\frac{}{\{q, q'\} = Q \setminus S} \quad (93)$$

$$\frac{}{Q^f = S^f} \quad (94)$$

$$\frac{}{q^0 = s^0}(s \neq s^0) \quad (95)$$

$$(s^0 \in S); ((q \xrightarrow{f_s(begin)} q' \in T));$$

$$\frac{(q' \xrightarrow{f_s(end)} s \in T)}{(q^0 = q)}(s = s^0) \tag{96}$$

$$\frac{(x \xrightarrow{l} x' \in E); (x' \neq s)}{(x \xrightarrow{l} x') \in T} \tag{97}$$

$$(\{q, q'\} = Q \setminus S);\ (\nexists q \xrightarrow{a} q' \in T \mid a \in L \cup \{f_s(begin), f_s(end)\});$$

$$\frac{(\nexists q' \xrightarrow{a} q \in T \mid a \in L \cup \{f_s(begin), f_s(end)\})}{(q \xrightarrow{f_s(begin)} q' \in T);\ (q' \xrightarrow{f_s(end)} s \in T)} \tag{98}$$

$$(\{q, q'\} = Q \setminus S);\ (x \xrightarrow{l} s \in E);$$

$$\frac{(q \xrightarrow{f_s(end)} q' \in T)}{(x \xrightarrow{a} q \in T)} \tag{99}$$

$Z := Alternative(X, Y)\ IN\ \{s\}$ [a]

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_s(begin), f_s(end)\}, T)$

---

[a] s is a constant that represents the state extension point.

$$\frac{}{(\{q, q'\} = Q \setminus S)} \tag{100}$$

$$\frac{}{(Q^f = S^f \cup q')} \tag{101}$$

$$\frac{}{(q^0 = s^0)} \tag{102}$$

$$\frac{(x \xrightarrow{l} x' \in E)}{(x \xrightarrow{l} x' \in T)} \tag{103}$$

$$\frac{(\{q,q'\} = Q \setminus S);\ (\nexists q \xrightarrow{a} q' \in T \mid a \in L \cup \{f_s(begin), f_s(end)\});\ (\nexists q' \xrightarrow{a} q \in T \mid a \in L \cup \{f_s(begin), f_s(end)\})}{(s \xrightarrow{f_s(begin)} q \in T);\ (q \xrightarrow{f_s(end)} q' \in T)} \tag{104}$$

$Z := Interrupt(X, Y)\ IN\ \{ALL\}$

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_{ep}(begin), f_{ep}(end)\}, T)$

$$\frac{}{(\{q,q'\} = Q \setminus S)} \tag{105}$$

$$\frac{}{(Q^f = S^f \cup q')} \tag{106}$$

$$\frac{}{(q^0 = s^0)} \tag{107}$$

$$\frac{(x \xrightarrow{l} x' \in E)}{(x \xrightarrow{l} x' \in T)} \tag{108}$$

$$\frac{(\{q,q'\} = Q \setminus S)}{(q \xrightarrow{f_{ep}(end)} q' \in T)} \tag{109}$$

$$\frac{(\{q,q'\} = Q \setminus S; (s \in S)); (q \xrightarrow{f_{ep}(end)} q' \in T)}{(s \xrightarrow{f_{ep}(begin)} q \in T)} \tag{110}$$

$Z := Graft(X, Y)\ IN\ \{(IN\ s_1,\ IN\ s_2)\}$ [a]

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_{ep}(begin), f_{ep}(end)\}, T)$

[a] s_1 is a constant that represents starting point of the referred UCEA and s_2 is the ending point such that $s_1 \neq s_2$.

$$\frac{}{(\{q\} = Q \setminus S)} \tag{111}$$

$$\frac{}{(Q^f = S^f)} \tag{112}$$

$$\frac{}{(q^0 = s^0)} \tag{113}$$

$$\frac{(x \xrightarrow{l} x' \in E)}{(x \xrightarrow{l} x' \in T)} \tag{114}$$

$$\frac{(\{q\} = Q \setminus S);\ (s_2 \in s)}{(q \xrightarrow{f_{ep}(end)} s_2 \in T)} \tag{115}$$

$$\frac{(\{q\} = Q \setminus S);\ (q \xrightarrow{f_{ep}(end)} s_2 \in T);\ (s_1 \in S)}{(s_1 \xrightarrow{f_{ep}(begin)} q \in T)} \tag{116}$$

Table 7: Builders Synthesis rules for UCA in the case of transition extension point with the qualifier BEFORE

$Z := Extend\_with(X, Y)\ BEFORE\ \{t = (s_1, a, s_2)\}$ [a]

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_t(begin), f_t(end)\}, T)$

[a] t is a constant that represents the transition extension point.

$$\frac{}{(\{q, q'\} = Q \setminus S)} \tag{117}$$

$$\frac{}{(Q^f = S^f)} \tag{118}$$

$$\frac{(x \xrightarrow{l} x' \in \{E \backslash t\})}{(x \xrightarrow{l} x' \in T)} \tag{119}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \xrightarrow{a} s_2 \in E);\ (q' \xrightarrow{f_t(begin)} q \notin T)}{(s_1 \xrightarrow{a} s_2 \in T);\ (s_1 \xrightarrow{f_t(begin)} q \in T);\ (q \xrightarrow{f_t(end)} q' \in T);\ (q' \xrightarrow{a} s_2 \in T)} \tag{120}$$

$Z := Alternative(X, Y)\ BEFORE\ \{t = (s_1, a, s_2)\}$ [a]

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_t(begin), f_t(end)\}, T)$

[a] t is a constant that represents the transition extension point.

$$\frac{}{\{q, q', q"\} = Q \setminus S} \tag{121}$$

$$\frac{(\{q, q', q"\} = Q \setminus S);\ (s_1 \xrightarrow{f_t(begin)} q \in T);\ (q \xrightarrow{f_t(end)} q' \in T);\ (q' \xrightarrow{a} q" \in T)}{(Q^f = S^f \cup \{q"\})} \tag{122}$$

$$\frac{(x \xrightarrow{l} x' \in \{E \backslash t\})}{(x \xrightarrow{l} x' \in T)} \tag{123}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \xrightarrow{a} s_2 \in E);\ (q' \xrightarrow{f_t(begin)} q \notin T)}{(s_1 \xrightarrow{a} s_2 \in T);\ (s_1 \xrightarrow{f_t(begin)} q \in T);\ (q \xrightarrow{f_t(end)} q' \in T);\ (q' \xrightarrow{a} q" \in T)} \tag{124}$$

$Z := Refine(X,Y)\ BEFORE\ \{t = (s_1, c_1, a, c_2, s_2)\}$ [a]

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_{ep}(begin), f_{ep}(end)\}, T)$

---

[a] t is a constant that represents the transition extension point.

$$\frac{}{(\{q\} = Q \setminus S)} \tag{125}$$

$$\frac{(\{q\} = Q \setminus S)}{(Q^f = S^f); (q^0 = s^0)} \tag{126}$$

$$\frac{(x \xrightarrow{l} x' \in \{E \setminus t\})}{(x \xrightarrow{l} x' \in T)} \tag{127}$$

$$\frac{(\{q\} = Q \setminus S);\ (s_1 \xrightarrow{a} s_2 \in E)}{(s_1 \xrightarrow{f_{ep}(begin)} q \in T);\ (q \xrightarrow{f_{ep}(end)} s_2 \in T)} \tag{128}$$

Table 8: Builders Synthesis rules for UCA in the case of transition extension point with the qualifier AFTER

$Z := Include(X,Y)\ AFTER\ \{t = (s_1, a, s_2)\}$ [a]

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_t(begin), f_t(end)\}, T)$

---

[a] t is a constant that represents the transition extension point.

$$\frac{}{(\{q, q'\} = Q \setminus S)} \tag{129}$$

$$\frac{}{Q^f = S^f} \tag{130}$$

$$\frac{(x \xrightarrow{l} x' \in \{E \backslash t\})}{(x \xrightarrow{l} x' \in T)} \tag{131}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \xrightarrow{a} s_2 \in E);\ (q' \xrightarrow{f_t(begin)} q \notin T)}{(q \xrightarrow{f_t(begin)} q' \in T);\ (q' \xrightarrow{f_t(end)} s_2 \in T);\ (s_1 \xrightarrow{a} q \in T)} \tag{132}$$

$Z := Extend\_with(X, Y)\ AFTER\ \{t = (s_1, a, s_2)\}^a$

$X = (S, s^0, S^f, L, E), X_b = (Q, q^0, Q^f, L \cup \{f_t(begin), f_t(end)\}, T)$

[a] t is a constant that represents the transition extension point.

$$\frac{}{(\{q, q'\} = Q \setminus S)} \tag{133}$$

$$\frac{}{(Q^f = S^f)} \tag{134}$$

$$\frac{(x \xrightarrow{l} x' \in \{E \backslash t\})}{(x \xrightarrow{l} x' \in T)} \tag{135}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \xrightarrow{a} s_2 \in E);\ (q' \xrightarrow{f_t(begin)} q \notin T)}{\begin{array}{r}(s_1 \xrightarrow{a} s_2 \in T);\ (s_1 \xrightarrow{a} q \in T);\ (q \xrightarrow{f_t(begin)} q' \in T);\\ (q' \xrightarrow{f_t(end)} s_2 \in T)\end{array}} \tag{136}$$

Figure 78 shows a set of base builders generated from UCA X with the consideration of a single extension point. Each builder is derived according to the semantics of the operator specified in the expression.

We draw the attention to two facts.

- First, in the case of *Alternative* operator and a transition t as extension point, we need to add three states, in contrast to the rest of the builders where we had to add only two states. This is due to the fact that with *Alternative* we have to duplicate the transition t and add two other transitions labeled with $begin_t$ and end_t, as shown in (f).

- Second, in the case of *Extend_with* operator associated with a state extension point x_1, as shown in (e), the outgoing transitions of x_1 have to be duplicated in order to preserve the set of traces of the original UCA. The additional behavior will form an alternative set of new traces to the outgoing transitions of x_1.

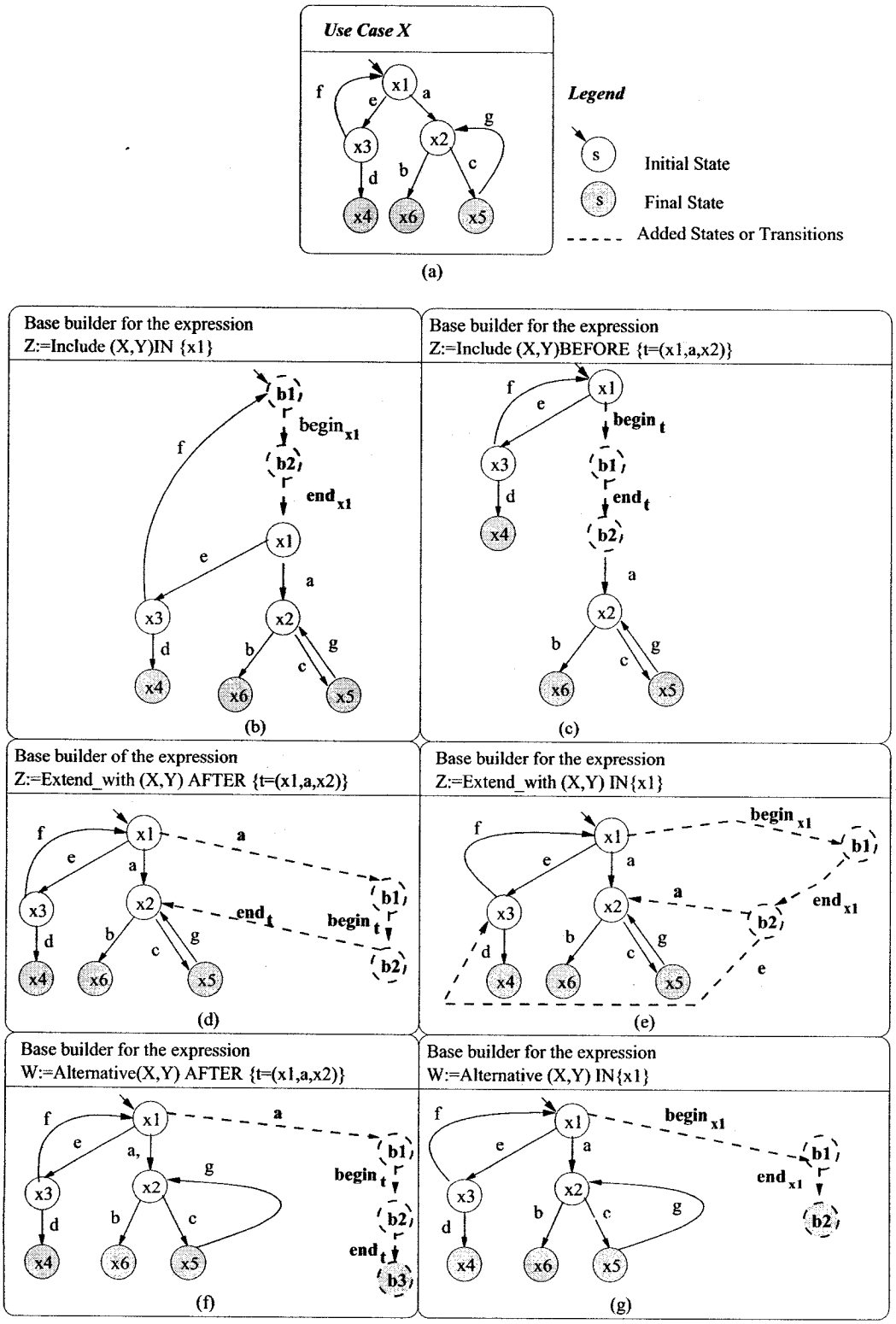


Figure 78: Examples Base Builders Generation

Appendix D

Final States Specification in the Case of Multiple Extension Points

Let's determine the set of final states S^f of the generated UCA from label matching in the case multiple extension points. Let A_1 and A_2 be the UCAs to compose. The final states are defined with respect to the composition operator specified between A_1 and A_2 and for the set of extension points. We distinguish four cases:

1. *Include, Graft, and Refine* ***Operator:***

   The set of final states of the new use case represents all the states labeled by one of the final states of the base use case, which are the final state of the base builder.

$$\frac{((s_1, s_2, ...s_n) \in S); (\forall s_i \in S_1, s_i \in S_1^f)}{((s_1, s_2, ...s_n) \in S^f)} \tag{137}$$

2. *Extend_with* ***composition operator: case of none of the final states of the base use case is an extension point:***

   The set of final states of the new use case represents all the states labeled by the final

states of the base use case, which are the final states of the base builder.

$$\frac{((s_1, s_2, ...s_n) \in S); (\forall s_i \in S_1, s_i \in S_1^f)}{(s_1, s_2, ...s_n) \in S^f} (\forall ep \in EP, ep \notin S_1^f) \tag{138}$$

3. *Extend_with* ***Operator: case of at least one of the extension points is a final state of the base use case***

In this case, the set of final states of the base use case are different from the final state of its builder. Hence, we will be using the set of final states of the generated base builders in order to determine the set of final states of the obtained UCA. Let $B_{1_i} = (Q_{1_i}, q_{1_i}^0, Q_{1_i}^f, L_{1_i}, T_{1_i}), 0 \leq i \leq m$ where m is the number of base builders generated from the base use case. Let $B_{2_i} = (Q_{2_i}, q_{2_i}^0, Q_{2_i}^f, L_{2_i}, T_{2_i}), 0 \leq i \leq m$ where m is the number of referred builders generated from the referred use case.

$$\frac{\begin{array}{c}((s_1, s_2, ...s_n) \in S); \\ (\forall s_i \in Q_{1_i}, s_i \in Q_{1_i}^f);\ (\forall s_i \in Q_{2_i}, s_i \in Q_{2_i}^f)\end{array}}{(s_1, s_2, ...s_n) \in S^f} (\exists ep \in EP, ep \in S_1^f) \tag{139}$$

The set of final states of the generated use case represents all the states labeled by the final states of the base builders and final states of the referred builders.

4. ***When*** *Alternative* ***and*** *Interrupt* ***operator:***

Let $B_{1_i} = (Q_{1_i}, q_{1_i}^0, Q_{1_i}^f, L_{1_i}, T_{1_i}), 0 \leq i \leq m$ where m is the number of base builders the set of generated builders from the base use case. Let $B_{2_i} = (Q_{2_i}, q_{2_i}^0, Q_{2_i}^f, L_{2_i}, T_{2_i}), 0 \leq i \leq m$ where m is the number of referred builders the set of generated builders from the referred use case. Therefore, the set of the final states in this case follows the rule:

$$\frac{((s_1, s_2, ...s_n) \in S); (\forall s_i \in Q_{1_i}, s_i \in Q^f_{1_i} \cup EP);\ (\forall s_i \in Q_{2_i}, s_i \in Q^f_{2_i})}{((s_1, s_2, ...s_n) \in S^f)} \quad (140)$$

We note that the fact that final states may also be specified as extension points does not affect the final states of the new UCA, an outcome related to the semantics of *Alternative*. We note also that with *Interrupt* operator, the set of extension points is the set of states of the base use case.

Appendix E

Rules of Builders Synthesis in the case of Use Case Extended Automata

Table 9: Builders Synthesis rules for UCEA in the case of state extension point

$Z := Include(X, Y)\ IN\ \{s\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$ [a]
$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$
[a] *Input_assgn* and *Out_assgn* are conditions. s is a constant that represents the extension point (state)

$$\frac{}{(\{q, q'\} = Q \setminus S)} \quad (141)$$

$$\frac{}{(Q^f = S^f)} \quad (142)$$

$$\frac{}{q^0 = s^0}(s \neq s^0) \tag{143}$$

$$\frac{(q \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q');\ (q' \overset{true,end,Out\_assign}{\longrightarrow} s)}{(q^0 = q)}(s = s^0) \tag{144}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in E);\ (x' \neq s)}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{145}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (\nexists q \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q' \in T);\ (\nexists q' \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T)}{(q \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q' \in T);\ (q' \overset{true,end,Out\_assign}{\longrightarrow} s \in T)} \tag{146}$$

$$\frac{(\{q, q'\} = Q \setminus S); (x \overset{c,l,a}{\longrightarrow} s \in E);\ (q \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q' \in T)}{(x \overset{c,l,a}{\longrightarrow} q \in T)} \tag{147}$$

$Z := Extend\_with(X, Y)\ IN\ \{s\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$ [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

---

[a] *Input_assgn* and *Out_assgn* are conditions. s is a constant that represents the state extension point.

$$\frac{}{\{q, q'\} = Q \setminus S} \tag{148}$$

$$\frac{}{Q^f = S^f}(s \notin S^f) \tag{149}$$

$$\frac{}{Q^f = S^f \cup \{q'\}}(s \in S^f) \tag{150}$$

$$\frac{}{q^0 = s^0} \tag{151}$$

$$\frac{(x \xrightarrow{c,l,a} x' \in E);\ (x \neq s)}{(x \xrightarrow{c,l,a} x' \in T)} \tag{152}$$

$$\frac{(\{q,q'\} = Q \setminus S);\ (s \xrightarrow{Cond\_comp,begin,Input\_assign} q \notin T);\ (q \xrightarrow{true,end,Output\_assign} q' \notin T)}{(s \xrightarrow{Cond\_comp,begin,Input\_assign} q \in T);\ (q \xrightarrow{true,end,Out\_assign} q' \in T)} \tag{153}$$

$$\frac{(\{q,q'\} = Q \setminus S);(x \xrightarrow{c,l,a} s \in E);\ (s \xrightarrow{Cond\_comp,begin,Input\_assign} q \in T);\ (q \xrightarrow{true,end,Output\_assign} q' \notin T)}{(s \xrightarrow{c,l,a} x' \in T);\ (q' \xrightarrow{c,l,a} x' \in T)} \tag{154}$$

$Z := Alternative(X,Y)\ IN\ \{s\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$ [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions. s is a constant that represents the state extension point.

$$\frac{}{(\{q,q'\} = Q \setminus S)} \tag{155}$$

$$\frac{}{(Q^f = S^f \cup q')} \tag{156}$$

$$\frac{}{(q^0 = s^0)} \tag{157}$$

$$\frac{(x \xrightarrow{c,l,a} x' \in E)}{(x \xrightarrow{c,l,a} x' \in T)} \tag{158}$$

$$\frac{(\{q,q'\} = Q \setminus S);\ (q' \xrightarrow{Cond\_comp,begin,Input\_assign} q \notin T)}{(s \xrightarrow{Cond\_comp,begin,Input\_assign} q \in T);\ (q \xrightarrow{true,end,Out\_assign} q' \in T)} \tag{159}$$

$Z := Interrupt(X,Y)\ IN\ \{ALL\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$ [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions.

$$\overline{\{q, q'\} = Q \setminus S} \tag{160}$$

$$\overline{(Q^f = S^f \cup q')} \tag{161}$$

$$\overline{(q^0 = s^0)} \tag{162}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in E)}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{163}$$

$$\frac{(\{q, q'\} = Q \setminus S)}{(q \overset{true,end,Out\_assign}{\longrightarrow} q' \in T)} \tag{164}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (q \overset{true,end,Out\_assign}{\longrightarrow} q' \in T);\ (s \in S)}{(s \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T)} \tag{165}$$

Table 10: Builders Synthesis rules for UCEA in the case of transition extension point with the qualifier BEFORE

$Z := Include(X,Y)$ $BEFORE$ $\{t = (s_1, c_1, a, c_2, s_2)\}$ $Cond\_comp$ $Input\_assgn$ Out_assgn [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions. t is a constant that represents the transition extension point.

$$\overline{\{q, q'\} = Q \setminus S} \tag{166}$$

$$\overline{Q^f = S^f; q^0 = s^0} \tag{167}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in \{E \setminus t\})}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{168}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in E);\ (q' \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \notin T)}{(s_1 \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T);\ (q \overset{true,end,Out\_assign}{\longrightarrow} q' \in T);\ (q' \overset{c_1,a,c_2}{\longrightarrow} s_2 \in T)} \tag{169}$$

$Z := Extend\_with(X,Y)$ $BEFORE$ $\{t = (s_1, c_1, a, c_2, s_2)\}$ $Cond\_comp$ $Input\_assgn$ Out_assgn [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions. t is a constant that represents the transition extension point.

$$\overline{\{q, q'\} = Q \setminus S} \tag{170}$$

$$\overline{Q^f = S^f; q^0 = s^0} \tag{171}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in \{E \backslash t\})}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{172}$$

$$\frac{(\{q,q'\} = Q \setminus S);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in E);\ (q' \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \notin T)}{(s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in T);\ (s_1 \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T);\ (q \overset{true,end,Out\_assign}{\longrightarrow} q' \in T);\ (q' \overset{c_1,a,c_2}{\longrightarrow} s_2 \in T)} \tag{173}$$

$Z := Alternative(X,Y)$ *BEFORE* $\{t = (s_1, c_1, a, c_2, s_2)\}$ *Cond_comp Input_assgn Out_assgn* [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions. t is a constant that represents the transition extension point.

$$\frac{}{\{q,q',q"\} = Q \setminus S} \tag{174}$$

$$\frac{(\{q,q',q"\} = Q \setminus S)}{(s_1 \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T);\ (q \overset{true,end,Out\_assign}{\longrightarrow} q' \in T);\ (q' \overset{c_1,a,c_2}{\longrightarrow} q" \in T \Rightarrow Q^f = S^f \cup \{q"\});\ (q^0 = s^0)} \tag{175}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in \{E \backslash t\})}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{176}$$

$$\frac{(\{q,q'\} = Q \setminus S);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in E);\ (q' \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \notin T)}{(s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in T);\ (s_1 \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T);\ (q \overset{true,end,Out\_assign}{\longrightarrow} q' \in T);\ (q' \overset{c_1,a,c_2}{\longrightarrow} q" \in T)} \tag{177}$$

$Z := Refine(X,Y)\ BEFORE\ \{t = (s_1, c_1, a, c_2, s_2)\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$ [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions. t is a constant that represents the transition extension point.

$$\frac{}{(\{q\} = Q \setminus S)} \tag{178}$$

$$\frac{}{(Q^f = S^f);\ (q^0 = s^0)} \tag{179}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in \{E \setminus t\})}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{180}$$

$$\frac{(\{q\} = Q \setminus S);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in E)}{(s_1 \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T);\ (q \overset{true,end,Out\_assign}{\longrightarrow} s_2 \in T)} \tag{181}$$

Table 11: Builders Synthesis rules for UCEA in the case of transition extension point with the qualifier AFTER

$Z := Include(X,Y)\ AFTER\ \{t = (s_1, c_1, a, c_2, s_2)\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$ [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions. t is a constant that represents the transition extension point.

$$\overline{\{q, q'\} = Q \setminus S} \tag{182}$$

$$\overline{Q^f = S^f; q^0 = s^0} \tag{183}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in \{E \backslash t\})}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{184}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in E);\ (q' \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \notin T)}{(q \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q' \in T);\ (q' \overset{true,end,Out\_assign}{\longrightarrow} s_2 \in T);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} q \in T)} \tag{185}$$

$Z := Extend\_with(X, Y)\ AFTER\ \{t = (s_1, c_1, a, c_2, s_2)\}\ Cond\_comp\ Input\_assgn\ Out\_assgn$ [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] *Input_assgn* and *Out_assgn* are conditions. t is a constant that represents the transition extension point.

$$\overline{\{q, q'\} = Q \setminus S} \tag{186}$$

$$\overline{Q^f = S^f; q^0 = s^0} \tag{187}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in \{E \backslash t\})}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{188}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in E);\ (q' \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \notin T)}{(s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in T);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} q \in T);\ (q \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q' \in T);\ (q' \overset{true,end,Out\_assign}{\longrightarrow} s_2 \in T)} \tag{189}$$

Z :=

$Alternative(X,Y)$ $AFTER$ $\{t = (s_1, c_1, a, c_2, s_2)\}$ $Cond\_comp$ $Input\_assgn$ Out_assgn [a]

$X = (S, s^0, S^f, L, V, I, E), X_b = (Q, q^0, Q^f, L \cup \{begin, end\}, V_b, I_b, T)$

[a] $Input\_assgn$ and Out_assgn are conditions. t is a constant that represents the transition extension point.

$$\overline{\{q, q', q"\} = Q \setminus S} \tag{190}$$

$$\frac{(\{q, q', q"\} = Q \setminus S);\ (s_1 \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \in T);\ (q \overset{true,end,Out\_assign}{\longrightarrow} q' \in T);\ (q' \overset{c_1,a,c_2}{\longrightarrow} q" \in T)}{(Q^f = S^f \cup \{q"\});\ (q^0 = s^0)} \tag{191}$$

$$\frac{(x \overset{c,l,a}{\rightarrow} x' \in \{E \backslash t\})}{(x \overset{c,l,a}{\rightarrow} x' \in T)} \tag{192}$$

$$\frac{(\{q, q'\} = Q \setminus S);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in E);\ (q' \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q \notin T)}{(s_1 \overset{c_1,a,c_2}{\longrightarrow} s_2 \in T);\ (s_1 \overset{c_1,a,c_2}{\longrightarrow} q \in T);\ (q \overset{Cond\_comp,begin,Input\_assign}{\longrightarrow} q' \in T);\ (q' \overset{true,end,Out\_assign}{\longrightarrow} q" \in T)} \tag{193}$$

Let's consider, for example, the case of *Include* operator in a state extension point. Rule 141 defines the set of states of the base builder as the set of states of the base use case increased with two states q and q'. Rule 142 defines the final state of the base builder as the final states of the base use case. Rule 143 defines the initial state of the base builder as the initial state of the base UCEA if the extension point is different from initial state of the

base UCEA. In the opposite case (the extension point is the initial state of the base UCEA), the initial state of the base builder is the added state q, as specified in Rule 144. Rule 145 implies that the builder evolves as the use case for all the transitions that are different from the ingoing transitions of the extension point s. According to Rule 146, a unique transition is specified between the two added states q and q'. This transition is labeled with *begin* and has as pre-condition the *Cond_comp* specified in the expression and as post-condition the *Input_assign* specified in the expression. Moreover, a unique transition is specified between the two states q' and s (s being the extension point). This transition is labeled with *end* and has as post condition the *Output_assign* specified in the expression. In addition, all outgoing transitions from s in the base UCEA are outgoing transitions from q' in the base builder. Rule 147 specifies that all the ingoing transitions to the state s in the base UCEA are ingoing transitions to q in the base builder.We notice that the set of variables of the generated base builder is obtained by the union of the set of variable of the base use case and the set of variables of the referred use case used in the specification of the composition condition, input assignment, and output assignment.

Bibliography

[1] OMG (2002) UML Resource Page, www.omg.org/uml/.

[2] *Message Sequence Chart (MSC). ITU Communication Standardization Sector (ITU-T. Z120 Recommendation for MSC-2000)*, 2000.

[3] http://www.agilemodeling.com/artifacts/usecasediagram.htm, Access Date: 04/2007.

[4] Telelogic AB: DOORS/ERS. http://www.telelogic.com/products/doorsers/, Access Date 04/2007.

[5] Use Case Map, http://www.usecasemaps.org/index.shtml, Access Date: 04/2007.

[6] http://javapathfinder.sourceforge.net/, Access Date July 2007.

[7] http://spinroot.com/spin/whatispin.html, Access Date July 2007.

[8] http://www.jflap.org/, Access Date July 2007.

[9] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[10] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *22nd International Conference on Software Engineering*, pages 304–313, 2000.

[11] R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Proc. 10th Intl. Conf. on Concurrency Theory*, pages 114–129. Springer Verlag, 1999.

[12] D. Amyot, D.Y. Cho, H. He, and Y. He. Generating Scenarios from Use Case Map Specifications. *Third International Conference on Quality Software (QSIC'03)*, pages 108–115, 2003.

[13] D. Amyot, L. Logrippo, and R. J. A. Buhr. Spécification et Conception de Systèmes Communicants: une Approche Rigoureuse Basée sur des Scénarios d'Usage. In *G. Leduc (Ed.), CFIP 97, Ingénierie des protocoles, Liège, Belgium*, pages 159–174, 1997.

[14] J. Araujo, W. Whittle, and D. Kim. Modeling and Composing Scenario-Based Requirements with Aspects . In *The 12th IEEE International Requirements Engineering Conference (RE 2004)*, Kyoto, Japan, September 2004.

[15] A. Arnold. *Finite Transition Systems.* 1994: Prentice-Hall.

[16] E. Baniassad and S. Clarke. Theme: An Approach for Aspect-Oriented Analysis and Design. *ICSE*, 0:158–167, 2004.

[17] F. Belina and D. Hogrefe. The CCITT-Specification and Description Language SDL . In *Computer Networks and ISDN Systems*, volume 16, pages 311 – 341, 1989.

[18] H. Ben-Abdallah and S. Leue. MESA: Support for Scenario-Based Design of Concurrent Systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 118–135, 1998.

[19] B. Bolognesi and E. Brinksma. Introduction to the OSI Specification Language LOTOS. In *Computer NEtworks and ISDN Systems*, volume 14, pages 25–59, 1987.

[20] Y. Bontemps and P. Heymans. Turning High-Level Live Sequence Charts into Automata. In *Proc. of "Scenarios and State-Machines: Models, Algorithms and Tools" (SCESM) Workshop of the 24th Int. Conf. on Software Engineering (ICSE 2002)*, Orlando, FL, May 2002. ACM.

[21] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language Users Guide.* Addison Wesley Longman, Inc. Reading, MA, 1999.

[22] F. Bordeleau. *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines.* PhD thesis, Department of Systems and Computer Engineering, Faculty of Engineering, Carleton University, Ottawa, Ontario, Canada, 1999.

[23] F. Bordeleau and J.-P. Corriveau. From Scenarios to Hierarchical State Machines: A Pattern-Based Approach. In *In Proceedings of OOPSLA 2000 Workshop on Scenario-based round-trip engineering*, Minneapolis, Minnesota, October 2000.

[24] F. Bordeleau and J.-P. Corriveau. On the Need for "State Machine Implementation" Design Patterns. In *Proceedings of ICSE 2002 Workshop on Scenarios and state machines: models, algorithms, and tools*, May 2002.

[25] S. Bourduas, F. Khendek, and D. Vincent. From MSC and UML to SDL. *In the Proceedings of IEEE Annual International Conference on Computer Software and Applications (COMPSAC'2002)*, Oxford, UK, August 26-29, 2002.

[26] R.J.A. Buhr. Use Case Maps: A New Model to Bridge the Gap Between Requirements and Detailed Design. *OOPSLA'95 Real Time Workshop*, October, 1995.

[27] R. W. Butler. What is Formal Methods?, 08/2001, Access Date 05/2007.

[28] E. M. Clarke and J. M. Wing. Formal methods: State of the Art and Future Directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[29] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[30] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir. Test Development for Communication Protocols: Towards Automation. In *Computer Networks, Volume 31, Issue 17*, pages 1835–1872, June 1999.

[31] H. Giese. Towards Scenario-Based Synthesis for Parametric Timed Automata. In *Proc. of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM), Portland, USA (ICSE 2003 Workshop 8)*, May 2003.

[32] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In *In Proceeding of the Fifth European Software Engineering Conference* , pages 254–271. Springer Verlag, 1995.

[33] M. Glinz. An Integrated Formal Model of Scenarios Based on Statecharts. In *In Schfer, W. and Botella, P. (eds.) (1995). Software Engineering - ESEC '95. Proceedings of the 5th European Software Engineering Conference, Sitges, Spain. Berlin, etc.: Springer (Lecture Notes in Computer Science 989).*, pages 254–271, 1995.

[34] M. Glinz, S. Berner, S. Joos, J. Ryser, N. Schett, and Y. Xia. The ADORA Approach to Object-Oriented Modeling of Software. In *Conference on Advanced Information Systems Engineering*, pages 76–92, 2001.

[35] O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. pages 94–101, 1994.

[36] H. Ben-Abdallah and S. Leue. Syntactic Detection of Process Divergence and Non-local Choice in Message Sequence Charts. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 259–274, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.

[37] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[38] D. Harel. From Play-In Scenarios to Code: An Achievable Dream. *Lecture Notes in Computer Science*, 1783:22+, 2000.

[39] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Lecture Notes in Computer Science*, 2088:1–33, 2001.

[40] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. *Proc. 4th International Conference On Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, 2002.

[41] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. *Lecture Notes in Computer Science*, 2154:39–58, 2001.

[42] Y. He, D. Amyot, and A. Williams. Synthesizing SDL from Use Case Maps: An Experiment. *In: 11th SDL Forum (SDL'03), LNCS*, 2708:117–136, 2003.

[43] G. J. Holzmann. Trends in Software Verification, 2003.

[44] G. J. Holzmann and M. H. Smith. Automating Software Feature Verification. *Bell Labs Technical Journal*, 5(2):72–87, - 2000.

[45] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* 3/E ed. 2007: Addison-Wesley.

[46] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal Approach to Scenario Analysis. In *IEEE Software*, volume 11, pages 33–41, 1994.

[47] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: a Use Case Driven Approach.* ACM Press; Wokingham, Eng.; Readings, Mass. Addison-Wesley, 1992.

[48] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development With Use Cases (Addison-Wesley Object Technology Series)* . Addison-Wesley , 2 edition, 2004.

[49] J. Kealey, K. Yongdae, D. Amyot, and G. Mussbacher. Integrating an Eclipse-Based Scenario Modeling Environment with a Requirements Management System. In *Electrical and Computer Engineering, Canadian Conference* , pages 2432–2435, May 2006.

[50] R. M. Keller. Formal Verification of Parallel Programs. *Comm. ACM 19*, (7):371–384, 1976.

[51] F. Khendek and G. Bochmann. Merging Behavior Specifications. *Formal Methods in System Design*, 6(3):259–293, 1995.

[52] F. Khendek and D. Vincent. Enriching SDL Specifications with MSCs, June 2000. In: 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000), Grenoble, France.

[53] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[54] T. Kishi and N. Noda. Analyzing Concerns used in Analysis/Design Techniques. *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001*, 2001.

[55] S. Kolahi. Composition and Verification of Behavioral Models. Master's thesis, Department of Computer Science, Concordia University, 2007.

[56] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A tool for dynamic modeling of object systems. Technical Report A-1996-4, Department of Computer Science, University of Tampere, Finland, 1996.

[57] K. Koskimies and E. Mäkinen. Automatic Synthesis of State Machines from Trace Diagrams. *Software - Practice and Experience*, 24(7):643–658, 1994.

[58] J. Koskinen, T. Männistö, and T. Systä. Minimally Adequate Synthesizer Tolerates Inaccurate Information during Behavioral Modeling. In *SCASE2001*, 2001.

[59] Kripke Structure Definition. *http : //en.wikipedia.org/wiki/kripke_structure*, Access Date 04/2007.

[60] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall. Upper Saddle River, NJ, 1998.

[61] W. J. Lee, S. Deok Cha, and Y. R. Kwon. Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering. *IEEE Trans. Softw. Eng.*, 24(12):1115–1130, 1998.

[62] S. Leue, L. Mehrmann, and M. Rezai. Synthesizing ROOM Models from Message Sequence Chart Specifications. *Technical Report 98-06, ECE Dept., University of Waterloo, Canada, April 1998. Short paper version in: 13th IEEE Conference on Automated Software Engineering, Honolulu, Hawaii*, October 1998.

[63] H. Liang, J. Dingel, and Z. Diskin. A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. *ICSE 2006 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools* , 2006.

[64] M. Lohrey. Realizability of High-Level Message Sequence Charts: Closing the Gaps. *Theoretical Computer Science*, 309(1):529–554, December 2003.

[65] LOTOS. http://wwwtios.cs.utwente.nl/lotos/, Access Date 04/2007.

[66] LTS Example. *http : //www.doc.ic.ac.uk/hf1/phd/thesis/html/index.html?a_complete_example.htm*, Access Date 04/2007.

[67] S. Mauw and M. A. Reniers. High-level Message Sequence Charts. In *Proceedings of the Eighth SDL Forum (SDL'97)*, pages 291–306, 1997.

[68] E. Mäkinen and T. Systä. MAS - An Interactive Synthesizer to Support Behavioral Modeling in UML. In *ICSE 2001, Toronto, Canada*, pages 15–24, 2001.

[69] R. Mizouni, A. Salah, and R. Dssouli. Interaction-Based Scenario Integration. In *Workshop on Model Design and Validation, ACM/IEEE Models/UML, Jamaica*, 2005.

[70] R. Mizouni, A. Salah, and R. Dssouli. Using Formal Composition of Use Cases in Requirements Engineering. In *The Nineteenth International Conference on Software Engineering and Knowledge Engineering SEKE'07, Boston, USA*, 2007.

[71] R. Mizouni, A. Salah, R. Dssouli, and S. Kolahi. Automated Approach for Use Case Composition. In *MCSEAI'06: 9th Maghrebian Conference on Information Technologies, Agadir, Morocco*, December 2006.

[72] R. Mizouni, A. Salah, R. Dssouli, and S. Kolahi. Incremental Extended Use Case Composition. In *7th International Conference on New Technol4ogies of Distributed Systems Morocco, Marrakesh*, June 4-8, 2007.

[73] R. Mizouni, A. Salah, R. Dssouli, and B. Parreaux. Integrating Use Cases with Explicit Loops. In *In Proceedings of Nouvelles TEchnnologies de la Rpartition (NOTERE'04), Saidia, Marocco*, June 2004.

[74] R. Mizouni, A. Salah, R. Dssouli, and B. Parreaux. Integrating Use Cases with Explicit Loops, June 2004.

[75] R. Mizouni, A. Salah, S. Kolahi, and R. Dssouli. Composition of Use Cases Using Synchronization and Model Checking. In E. Najm, J. F. Pradat-Peyre, and V. Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2006.

[76] R. Mizouni, A. Salah, S. Kolahi, and R. Dssouli. Roles of variables in Use Case Composition. In *New Technologies for Distributed Systems (NOTERE'2006), Toulouse, France* , 2006.

[77] R. Mizouni, A. Salah, S. Kolahi, and R. Dssouli. Composition of partial system behaviors. *IET Software journal (formerly IEE Proceedings, Software)*, 1:143–160, 2007.

[78] H. Muccini. An Approach for Detecting Implied Scenarios. In *Proc. ICSE 2002 Workshop on "Scenarios and State Machines: Models, Algorithms, and Tools"*, May 2002.

[79] H. Muccini. Detecting Implied Scenarios Analyzing Non-local Branching Choices . In *Proc. Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2003), ETAPS2003, Warsaw, Poland, April 2003. LNCS.*, 2003.

[80] A. Muscholl and D. Peled. Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces. In Tomasz Wierzbicki Miroslaw Kutylowski, Leszek Pacholski, editor, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings.*, volume 1672. Springer, 1999.

[81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1981.

[82] D. C. Petriu, D. Amyot, and C. Murray Woodside. Scenario-Based Performance Engineering with UCMNAV. In R. Reed and J. Reed, editors, *SDL Forum*, volume 2708 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2003.

[83] A. Rashid, A. Moreira, and A. Arajo. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *Requirements Engineering*, pages 199–202, 2002.

[84] A. Rashid, A. Moreira, and A. Arajo. Modularisation and Composition of Aspectual Requirements. In *Aspect Oriented Software Development Conference*, pages 11–20, 2003.

[85] W. Reisig. Petri Nets: An Introduction. In *EATCS Monographs on Theoretical Computer Science, Volume 4. Springer Verlag*, 1985.

[86] G. Robert, F. Khendek, and P. Grogono. Deriving an SDL Specification with a Given Architecture from a Set of MSCs. In *A. Cavalli and A. Sarma (editors), SDL'97 : Time for Testing - SDL, MSC and Trends, Proceedings of the eight SDL Forum*, 1997.

[87] R. F. Roggio. Use Cases and Traceability: a Marriage for Improved Software Quality. In *16th Annual NACCQ, Palmerston North, New Zealand, July, 2003 (eds) Mann, S. and Williamson, A.*, 2003.

[88] J. Ryser and M. Glinz. Dependency Charts as a Means to Model Inter-Scenario Dependencies. In *In G. Engels, A. Oberweis and A. Zndorf (eds.): Modellierung 2001. GI-Workshop, Bad Lippspringe, Germany. GI-Edition - Lecture Notes in Informatics*, volume P-1, pages 71–80, 2001.

[89] S. Uchitel and J. Kramer and J. Magee. Behavior Model Elaboration using Partial Labeled Transition Systems . *ESEC/FSE 2003*, 2003.

[90] M. Sabetzadeh and S. Easterbrook. View Merging in the Presence of Incompleteness and Inconsistency. *Requirements Engineering*, 11(3):174–193, 2006.

[91] A. Salah. *Génération Automatique d'une Spécification Formelle à de Scénarios Temps reél.* PhD thesis, Univesité de Montréal, Faculté des études supérieures, 2002.

[92] A. Salah, R. Dssouli, and G. Lapalme. Implicit Integration of Scenarios into a Reduced Timed Automata. *Information and Software Technology*, 45, Issue 11:715–725, 2003.

[93] SCED tool. http://www.cs.tut.fi/ tsysta/sced/, Access Date 04/2007.

[94] B. Sengupta and R. Cleaveland. Triggered Message Sequence Charts. In ACM Press, editor, *In SIGSOFT2002/FSE-10, Charleston, SC, USA* , November 2002.

[95] S. Somé. *Dérivation de Spécification à partir de Scénarios d'Interaction* . PhD thesis, Univesité de Montréal, Faculté des études supérieures, June 1997.

[96] S. Somé. Beyond Scenarios: Generating State Models from Use Cases. In *Proceedings of ICSE 2002 Workshop on Scenarios and state machines: models, algorithms, and tools*, 2002.

[97] S. Somé and R. Dssouli. An Enhancement of Timed Automata generation from Timed Scenarios using Grouped States. *Electronic journal on Network and Distributed Processing*, 6, 1998.

[98] S. Somé, R. Dssouli, and J. Vaucher. From Scenarios to Timed Automata: Building Specifications from Users Requirements, 1995.

[99] T. Systä, K. Koskimies, and E. Mäkinen. Automated Compression of State Machines Using UML Statechart Diagram Notation. In *Information and Software Technology*, volume 44, pages 565–578, 2002.

[100] J. Kealeyand E. Tremblay, J.-P. Daigle, J. McManusand O. Clift-Nol, and D. Amyot. jUCMNav: New platform for the edition an the analysis of UCM . *K. Adi, D. Amyot and L. Logrippo (editors) Proceeding of the 5th conference of new Distributed technology (les Nouvelles Technologies de la Rpartition NOTERE 2005, Gatineau, Canada)*, pages 215–222, 2005.

[101] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In *Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[102] S. Uchitel and M. Chechik. Merging Partial Behavior Models. *SIGSOFT'04/FSE* , 2004.

[103] S. Uchitel, J. Kramer, and J. Magee. Detecting Implied Scenarios in Message Sequence Chart Specifications. In *In proceedings of the 9th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'01). Vienna, Austria. September 2001*, 2001.

[104] S. Uchitel, J. Kramer, and J. Magee. Modeling Undefined Behavior in Scenario Synthesis. In *ICSE'03*, 2003.

[105] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. In *IEEE Transactions on Software Engineering*, volume 29:2, February 2003.

[106] J. Whittle. *The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor.* PhD thesis, University of Edinburgh, 1998.

[107] J. Whittle and J. Arajo. Scenario Modeling with Aspects. *IEE Proceedings - Software*, 151(4):157–172, 2004.

[108] J. Whittle, R. Kwan, and J. Saboo. From Scenarios to Code: An Air Traffic Control Case Study . In *International Conference on Software Engineering (ICSE2003), Portland, Orego*, 2003.

[109] J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *International Conference on Software Engineering*, pages 314–323, 2000.

[110] J. Whittle and J. Schumann. Statechart Synthesis From Scenarios: An Air Traffic Control Case Study . In *International Conference on Software Engineering (ICSE2002)*, 2002.

[111] R. Wieringa. A survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998.

[112] R. Young. *The Requirements Engineering Handbook.* Artech House, 2004.

[113] T. Zheng and F. Khendek. An extension to MSC-2000 and its application. In *Proceedings of the International Workshop on SDL and MSC (SAM), Aberystwyth, Wales*, volume Lecture Notes in Computer Science 2599. Springer Verlag, June 2002.

[114] P. Zielczynski. Traceability from Use Cases to Test Cases, http://www-128.ibm.com/developerworks/rational/library/04/r-3217/, Access Date: 04/2007.