FORMAL SEMANTICS AND VERIFICATION OF USE CASE MAPS

Jameleddine Hassine

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

April 2008

Your file Votre référence
ISBN: 978-0-494-37746-8
Our file Notre référence
ISBN: 978-0-494-37746-8

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By:	**Mr. Jameleddine Hassine**
Entitled:	**Formal Semantics and Verification of Use Case Maps**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_______________ Chair
_______________ External Examiner
_______________ Examiner
_______________ Examiner
_______________ Examiner
_______________ Supervisor
_______________ Co-supervisor

Approved _______________
Chair of Department or Graduate Program Director

__________ 20 ____ _______________
Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

Abstract

Formal Semantics and Verification of Use Case Maps

Jameleddine Hassine, Ph.D.
Concordia University, 2008

Common to most software development processes is that system functionalities are defined early in the life cycle in terms of informal requirements and visual models. As requirement descriptions evolve, they quickly become error-prone and difficult to understand leading to prolonged detrimental effects on reliability, cost, and safety of a software system that are very costly to fix in later phases of the software development process. Thus, the development of techniques and tools to support requirement specification development, understanding, validation, verification, maintenance and reuse becomes an important issue.

This thesis proposes a novel methodology named *Early Stages V&V* (Early Stages Validation & Verification), which combines the semi-formal scenario-based Use Case Maps language with formal techniques to help comprehend, validate and verify requirements. UCM models allow the description of functional requirements and high-level designs at early stages of the development process. Use Case Maps is being standardized as part of the User Requirements Notation (URN), the most recent addition to ITU-Ts family of languages. In the first part of the thesis, we propose a concise and rigorous formal semantics for Use Case Maps based on Abstract State Machines (ASM) formalism. The resulting semantics are embedded in an ASM-UCM simulation engine and are expressed in AsmL, an advanced ASM-based executable specification language, which is used to validate UCM models through simulation.

Timing issues are often overlooked during the initial system design and treated as separate behavioral issues and therefore described in separate models. In the second part of the thesis, we extend the Use Case Maps language to cover timing constraints. A potential timed version of UCM (called *Timed UCM*) is formalized using Clocked Transition Systems (CTS) and Timed Automata (TA). The proposed semantics can be applied to comprehend, analyze, validate and verify (using model checking) timed UCM models. In addition, we have proposed a novel UCM-based property pattern system that combines qualitative, real-time and architectural properties into a single graphical representation. The resulting pattern system is mapped to popular temporal logics CTL, TCTL and ArTCTL (Architectural real-time temporal logic), which is an extension to TCTL introduced in this research that provides temporal logics with architectural scopes.

In order to achieve an efficient validation and verification of UCM models and to assess the impact of a specification change (e.g. as a result of a bug fixing or a feature upgrade), we extend the application of the well-known technique of program slicing to Use Case Maps language.

An ongoing example of a simple telephone system is used to illustrate these concepts. The thesis validates the *Early Stage V&V* methodology by implementing it and applying it to two case studies: IP Multicast Protocol and an Online Store application.

Acknowledgments

First and foremost, I would like to express my most sincere gratitude to my supervisors, Professor Juergen Rilling and Professor Rachida Dssouli, without whose insightful guidance and invaluable help this thesis would not be possible. They have given me enormous freedom to pursue my own interests and provided me with guidance and steadfast encouragement to ensure that my efforts are rewarded. I really appreciate the time and energy they had dedicated to my thesis work. I feel fortunate to be under their supervision.
I would like to give great thanks to professor Daniel Amyot, School of Information Technology and Engineering at the University of Ottawa, who also played an important role in my research work. I am grateful for his guidance throughout this research. As my thesis reader, his insightful suggestions were invaluable in the production of this work.

I also extend my thanks to my committee, Dr. Abdeslam En-Nouaary, Dr. Dhrubajyoti Goswami, Dr. Hon F. Li, and Dr. Abdellatif Obaid (From UQAM), my external examiner, for gracefully accepting to review and comment this work.

Finally, words cannot express the thanks I owe to my parents and my wife for their encouragement and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The hardest single part of building a software system is deciding what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No part of the work so cripples the resulting systems if done wrong. No other part is more difficult to rectify later. Jr. Frederick P. Brooks [FPB87]

1.1 Motivation

There is a general consensus on the importance of good *Requirements Engineering (RE)* for achieving high quality software. The modeling and analysis of requirements have been the main challenges during the development of complex systems. Requirements describe the needs or desired functionalities of a product, i.e. what the system should do. Requirement has many definitions, each emphasizing an aspect of requirements engineering [ZJ97]. Analysts categorize requirements into the following types (for a more refined description, see [Poh96]):

- **System.** These requirements describe the type of system, such as hardware or software. They may describe requirements concerning the development process (cost-effective, timely) or development aspects of the resulting product (reusable, maintainable, platform independent).

- **Functional and non-functional.** These requirements describe the form of service. Functional requirements are associated with specific functions, tasks or behaviors the system must support. They describe a service relation between inputs and outputs. Non-functional requirements do not define a service, but instead describe constraints on various attributes of the service provision, such as efficiency and reliability. Non-functional requirements are sometimes called system qualities.

- **Abstraction level.** Analysts describe requirements at different levels of abstraction. By specializing or refining abstract requirements, or by generalizing detailed requirements, they define a requirements abstraction hierarchy.

- **Representation.** One requirement can have several representations. It may begin as an informal sketch, become a natural language document, and end up as a more formal representation (e.g., temporal logic [Pnu77, MP92])

The process of defining requirements is called *requirement engineering.* It constitutes the first phase of any development process and it covers all the activities involved in discovering, documenting and maintaining a set of requirements for a computer-based system [SS97]. A requirement engineering process is expected to include four activities: requirements elicitation where the customers' needs are identified through consultation with all stakeholders; requirements analysis which, based on the customer's context and constraints, checks and solves potential conflicts, overlaps, omissions and inconsistencies; requirements specification that describes formally or informally the behavior of the system to be developed; and requirements validation which involves verifying if a specification is complete and clear enough for the development team to understand exactly what it needs to build.

Although several approaches have been suggested to address these tasks, requirements engineering is still facing many challenges. Among many others, we cite:

- Lack of requirement languages for eliciting and describing requirements according to their nature (e.g. functional, non-functional), and their level of abstraction. Existing frameworks tend to be overloaded with linguistic constructs and irrelevant details.

- There is a gap between requirement engineering research and formal methods research. Traditionally, requirement languages suffer from a lack of formal semantics. Hence, this represents a barrier towards the adaptation of rich set of analysis methodologies offered by formal methods.

- Lack of automated analysis and validation support to detect possible ambiguities, inconsistencies or undesirable interactions.

- Lack of specialized support: Several aspects of the requirements engineering process need specialized support for requirements evolution and refinement.

Scenario-based Models

Functional requirements capture the intended behavior of a system. This behavior may be expressed as services, tasks, functions a system is required to perform or as transformations from inputs to outputs. Many models (Prototype model, Use Case Model, Organized by Roles Model, etc.) have been proposed for capturing user requirements. The most commonly used model is *use cases.* There are several reasons why use cases have become popular and universally adopted. According to [JBR99] the two major reasons are:

- They offer systematic and intuitive means of capturing functional requirements and uncovering hidden requirements.

- They drive the whole development process since most activities such as analysis, design, validation and verification are performed starting from use cases.

Although semi-formal, scenario driven approaches have raised a higher level of interest and acceptance mostly because of their intuitive representation. Scenarios are known to help describing functional requirements, uncovering hidden requirements and trade-offs, as well as validating and verifying requirements. Scenarios can also be applied to requirements to support different development stages, including user requirements, system requirements, and testing requirements. Lamsweerde [Lam00] provides a thorough discussion on the relationships between scenarios and other requirements models. To avoid an explosion in the number of individual scenarios describing a complex system, several approaches have been developed. These approaches capture common scenario parts (often called episodes) and describe interdependencies through relationships such as precedence, alternatives, inclusion, extension, usage, etc., while at the same time improving consistency and maintainability. Breitman et al. [BdP00] provided an extensive case study on scenario evolution based on such relationships, and they proposed a taxonomy for classification and heuristics for the identification of scenario relationships. The authors in [HO03] presented an integration method to detect the inconsistency between scenarios from different viewpoints and to provide support for scenario evolution by generating new scenarios from an integrated scenario.

In this thesis, we focus on functional requirements described using scenario notations [AE03], more specifically Use Case Maps notation [IT02b] that is part of the User Requirements Notation (URN), the most recent addition to ITU-T's family of languages. UCMs are used to capture and analyze system behavior at an abstraction level that is above both inter-component communication and detailed level component behavior. They describe multiple scenarios in a single, integrated view, as well as the relationships between scenarios and their underlying architecture. This promotes the understanding and reasoning about the system as a whole, as well as focusing on individual scenario description, scenario interaction, and responsibility allocation, before introducing inter-component communication. UCMs have been successful in describing and validating a wide range of systems, including Wireless Intelligent Networks [AA99, Yi00], Wireless ATM [And00], GPRS [DAF98], agent systems [BEGM98], and Web applications [ARW05]. They have been used in other types of applications such as program comprehension [AMM02].

Recognizing the need to incorporate non-functional aspects, and in particular time-related aspects into requirement languages in order to correctly model time dependent applications at early stages during system development, we extend Use Case Maps language with notion of time. This can support quantitative analysis at early phases of the software development process. Thus, help detecting design errors early and reduce the cost of later redesign activities when it turns out that time constraints, for instance, are not met.

Early Error Detection

A major motive for spending time and effort on requirements engineering and its improvement comes from the objective of doing the software development right from the beginning, instead of patching at the end. This objective is justified by the empirical evidence supporting the following hypotheses [Dav93].

- Many requirements errors are being made.

- Many of theses errors are detected late.

- Many of these errors can be detected early in the life cycle.

- Not detecting these errors may contribute to dramatic increase of software costs.

In [Dav93], it has been shown that the cost of detecting and repairing errors increases dramatically as the development process proceeds. Table 1.1 shows a compilation of three empirical studies, indicating that it may be up to 200 times more expensive to detect and repair errors in the operation stage, compared to detecting and repairing them during the requirements stage [Dav93].

Stage	**Relative cost of error repair**
Requirements	0.1-0.2
Design	0.5
Implementation	1
Component Verification	2
System Validation	5
Operation	20

Table 1.1: Relative Cost of Error Repair in Different Development Stages [Dav93]

With these figures in mind, it is reasonable to believe that the highest risk is related to the requirements specification phase of the development process. Furthermore, many of the efforts in other stages of the system development lifecycle, for instance system test, depend on the correctness of the requirements specification. Formal methods are very helpful at finding errors early on and can nearly eliminate certain classes of error [Hal90].

Formal Semantics

Many authors [NE00, Hal90] have identified the need to move from contextual enquiry to elicit requirements, to more formal representations for analysis. Creating a formal specification forces the user to make a detailed system analysis that usually reveals errors and inconsistencies in the informal requirement specification [Som06]. The main motivation for requirement formalization is the need for users to have a common and more precise understanding of the requirement and to remove any existing ambiguity. The lack of accuracy in the definition of a requirement language can cause problems regarding the models expressed in the language, such us different interpretations, etc. A requirement specification can be made unambiguous and clear by attaching a formal, mathematical semantics to it (called formal semantics). Moreover, this need for formal semantics comes from a desire to have better ways to verify properties of specifications, as well as to provide better means to check the correctness of specifications [Hal90]. A formal description can be used to formulate equivalence relations (semantical equivalence) between specifications. Equivalence relations defined on models may abstract specifications from details, and provide satisfactory notions of semantics equivalence and implementation correctness.

Validation and Verification

As requirements evolve, they quickly become error-prone and difficult to understand. The discovery of errors in early development stages significantly reduces development time and cost. The use of validation and verification (V&V) techniques increases our degree of assurance that the final product meets user expectations and satisfies the given specification. V&V is usually applied to a product or to a model. Reviews, inspections, simulation, walkthroughs and testing represent the major V&V techniques.

Validation answers the question: *Are we building the right product?*. Validation is an activity that ensures the correctness of the final product with respect to the stakeholders' true needs and expectations (requirements).

Verification answers the question: *Are we building the system right?*. Verification is an activity that ensures that the selected design solution satisfies the specification, and that the end product satisfies the design. Ultimately, verification is the process of determining whether a system satisfies a given property of interest.

While simulation and testing (validation techniques) explore some of the possible behaviors and scenarios of a system, they leave open the question of whether the unexplored trajectories may contain fatal bugs. Formal verification conducts an exhaustive exploration of all possible behaviors [EMCGP99] and will be able to discover these bugs.

Today the best known verification methods are model checking [EMCGP99] and theorem proving [Duf91, CLL97], both of which have sophisticated tool support and have been applied to non-trivial systems. Model checking [EMCGP99] is a technique to automatically verify functional requirements of behavioral models. The functional requirements are specified in temporal logic [Pnu77, MP92]. Model checkers (tools implementing model checking) verify a functional requirement against a specific property by searching the complete state space of the behavioral model. If the model checker does not find an error, the property is certain to hold. If the model checker does find an error, the model checker returns a counterexample in the form of a sequence of states that violates the property. This feedback of the model checker can help the modeler in finding the error and repairing it. The other formal verification technique is theorem proving [CW96]. Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system.

Advantage of model checking over theorem proving is that with model checking, user requirements can be verified automatically without any user interaction, whereas with theorem proving, user interaction may be required. Also, if the functional requirement fails to hold, the model checker returns a counterexample whereas theorem provers do not do this. The major disadvantage of model checking is that it is only suitable for finite state spaces, whereas theorem proving can handle both finite and infinite state spaces. The state space of the model must be finite, since model checking requires an exhaustive search on the model state space. In Section 8.1.3, we discuss some technique to tackle the state space explosion problem.

The broader goal of this thesis is to provide techniques to help improve requirement specification quality, through formal prototyping and validation. More specifically, this thesis presents a methodology that applies proven concepts and techniques in order to describe and validate requirement specifications at the early stages of system development.

1.2 Research Hypothesis

The past twenty-five years have seen the advent of many different requirement engineering techniques to improve the quality of requirements. We can distinguish two classes of techniques: informal, techniques (e.g., OMT [RBL+90], BOOCH [CC96]) which emphasize ease-of-use and comprehension, often at the cost of rigor and reliability, and the formal specification techniques (FSTs) (e.g., SDL [IT02a], LOTOS [ISO89]) which emphasize formality, often at the cost of ease-of-use and understandability. It has been suggested that no single method for software development is a panacea [Jac95]. That is, there is no individual method that will meet all of the challenges presented in the previous section. Combining formal and informal methods is a research subject where much work has been done [Hal96, Bor99, UKM03, Amy01a, Mau96, NN92, EW01].

In this thesis, we present an innovative approach, where we combine formal specification techniques (FSTs) with the semi-formal language Use Case Maps (UCMs) [IT02b]. We selected Abstract State Machines (ASM) [BS03] as our FST. ASMs use classical mathematical structures to model any algorithm at its natural abstraction level. The ASM authors claim that ASMs have the following desirable characteristics: Precision, Faithfulness, Understandability, Executability, Scalability, Generality. ASMs can be used in a wide variety of domains (e,g., sequential, parallel, and distributed systems; abstract-time and real-time systems; finite-state and infinite-state domains) and can describe systems at several different layers of abstraction. Hence, ASM is an appropriate formalism for the description of functional requirements at a high level of abstraction (i.e., at the Use Case Maps abstraction level).

Research Hypothesis 1

Our first research hypothesis is denoted as follows:
At the early stages of system development, requirements described using the Use Case Map language can be formalized in terms of Abstract State Machines (ASM). Hence, UCM models can be validated through simulation and functional testing.

Research Hypothesis 2

Although much work has been done in the model-based verification methodologies [QS82, CES86, Var91], many challenges such as the resolution of state-space explosion problem [McM92, Val91, CGL92, CFJ93] still remain open research subjects. Reduction techniques have been used to solve the state space explosion problem [MT98]. We believe that the use of reduction techniques [Wei84] at the Use Case Maps abstraction level can help reduce the specification size allowing for a more efficient validation and verification of requirements.

Our second research hypothesis is denoted as follows:
In the process of verifying complex systems, requirements described using the Use Case Map language can be validated and verified efficiently through the use of reduction techniques.

Research Hypothesis 3

UCM models focus on the description of functional and behavioral requirements, as well as, high-level designs at the early stages of system development processes. However, timing issues are often overlooked during the initial system design. They are typically regarded as separate behavioral issues and therefore described in separate models. We believe that timing aspects must be integrated into the system model at an early development stage, to allow for a consistent analysis throughout all life-cycle phases of software product. We believe also that Use Case Maps notation can be extended to cover non-functional requirements such as timing constraints. A potential timed version of UCM (called *timed UCM*) can be formalized using *Clocked Transition System* (CTS) and *Timed Automata* (TA) [AD94]. The theory of Clocked Transition Systems and Timed Automata provide a formal framework to model and to analyze the behavior of real-time systems, that is, of systems whose correct functioning is subject to and must ensure the respect of strict timing constraints such as execution times, response times and so on.

Our third research hypothesis is denoted as follows:
Use Case Maps notation can be extended to cover non-functional requirements such as timing constraints. Timed UCM can be formalized in terms of CTS (Clocked Transition System) and Timed Automata (TA) formalisms that can be analyzed and verified.

Research Hypothesis 4

Although there exists a significant body of research in the area of formal verification and model checking tools for software and hardware systems, there has been so far only a limited industry and end-user acceptance of these tools. Beside the technical problem of state space explosion, one of the main reasons for this limited acceptance is the unfamiliarity of users with the required specification notation. Requirements have to be typically expressed as temporal logic formalisms and notations. Property specification patterns [DAC99, GL05, KC05a, KC05b, MP92, NV90] were successfully introduced to bridge this gap between users and model checking tools. They enable also non-experts to write formal specifications that can be used for automatic model checking.

We believe that Use Case Maps can be used to describe a set of commonly used properties that are presented in terms of occurrence, ordering and temporal scopes of actions. Furthermore, UCM also supports the description of properties with respect to their architectural scope. This may be achieved through a minimal extension of UCM language.

Our fourth research hypothesis is denoted as follows:
The visual and easy to learn syntax of UCM, can support the description of a large set of high level properties without the need for temporal logic formalisms.

1.3 Thesis Approach

UCMs have been successfully used in describing real-time systems, with a particular focus on telecommunication system and services [AA99, ALBG99, Amy01a] as well as the description of business process models [ARW05]. We believe that they fit well in the Early Stages V&V approach proposed in this thesis. We intend to validate the research hypothesis by developing *Early stages V&V approach* and by successfully applying it to various applications.

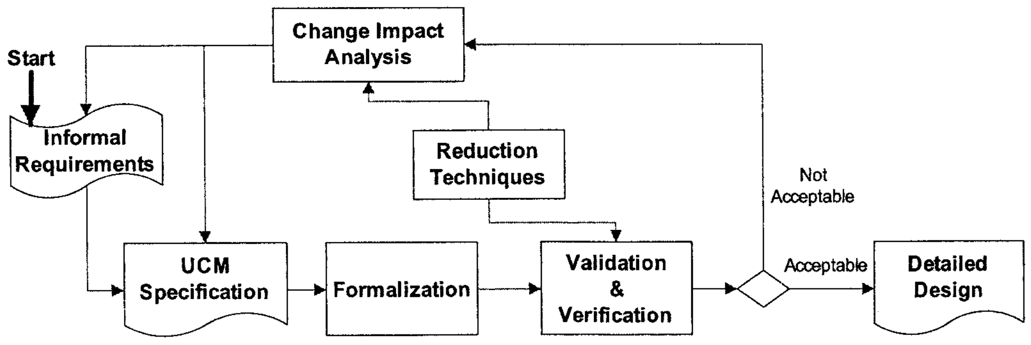


Figure 1.1: Early Stages Validation and Verification Approach

We adopt an iterative approach with a sequence of iterations. Each iteration includes all the building blocks of our V&V approach, as shown in Figure 1.1. System functionalities, architecture and timing constraints are captured as Use Case Maps scenarios (requirements elicitation). A UCM model can be constructed based on informal requirements or use cases [Jac04], where separate UCMs can be created for individual scenarios and integrated later to have a global UCM specification.

UCM system specifications are then formalized in terms of ASM and/or TA formalisms, allowing for formal validation and verification. If logical and design errors are detected, a change impact analysis is performed in order to locate which parts of the specification would be impacted by the proposed fix. This process leads to a modification on both the informal requirements and its corresponding UCM specification. If no problem is detected, then the specification is declared to be error free and may be refined to a more detailed system design model.

1.4 Thesis Contributions

This thesis offers four main contributions:

1.4.1 Contribution 1: Early Stages V&V methodology

Different theories and techniques are involved in the support of the Early Stages V&V methodology. Some of them, such as Slicing [Wei84] and Model Checking [EMCGP99], already exist. However, the proposed approach extends their application. The following published papers describe some of the used techniques.

1. **An ASM Operational Semantics of Use Case Maps (IEEE International Conference on Requirements Engineering - RE 2005) [HRD05b], Abstract Operational Semantics for Use Case Maps (Formal Techniques for Networked and Distributed Systems - FORTE 2005,) [HRD05a].**
   These two papers propose a formal operational semantics for Use Case Maps language based on Multi-Agent Abstract State Machines. The ASM model provides a concise semantics of UCM functional constructs and describes precisely the control semantics. The resulting operational semantics are embedded in an ASM-UCM simulation engine and are expressed in *AsmL*, an advanced ASM-based executable specification language. The proposed ASM-UCM engine provides an environment for executing and simulating UCM specifications.

2. **Applying Reduction Techniques to Software Functional Requirement Specifications (System Analysis and Modeling- SAM 2004) [HDR04].**
   This paper extends the well-known technique of program slicing to Functional Requirement Specification based on Use Case Maps notation. This new application of slicing, called *UCM Requirement Slicing* is useful to aid requirement comprehension, validation, verification and maintenance.

3. **Change Impact Analysis with Use Case Maps (IEEE International Workshop on Principles of Software Evolution - IWPSE 2005) [HJRD05].**
   This paper presents a novel approach to change impact analysis at the requirement level. Both slicing and dependency analysis at the Use Case Map specification level are used to identify the potential impact of requirement changes on the overall system.

4. **Modification Analysis Support at the Requirements Level (IEEE International Workshop on Principles of Software Evolution - IWPSE 2007) [SHR07].**
   This paper presents a novel approach that combines UCM with Formal Concept Analysis (FCA) to assist decision makers in supporting modification analysis at the requirements level. The proposed approach provides support for determining the potential modification and re-testing effort associated with a change without the need to analyze or comprehend source code.

5. **Timed Use Case Maps (System Analysis and Modeling: Language Profiles - SAM 2006) [HRD06].**
   This paper introduces an approach to describe timing constraints in Use Case Maps specifications. It provides a formal semantics of Timed UCM in terms of Clocked Transition Systems (CTS) over a discrete model of time.

6. **Formal Verification of Use Case Maps with Real Time extensions (SDL Forum - SDL 2007) [HRD07a].**
   This paper presents a formal operational semantics of Timed UCM in terms of Timed Automata (TA) that can be analyzed and verified with the UPPAAL model checker tool.

1.4.2 Contribution 2: Classification of Timed Scenario Languages

The need to incorporate non-functional aspects, and in particular time-related aspects into requirement languages has been widely recognized. This is essential in order to correctly model time dependent applications at early stages in system development. Typical examples of such applications are communication protocols and real-time distributed systems. In this thesis, eleven evaluation criteria were proposed to classify and compare thirteen timed scenario languages. Chapter 6 presents and discusses the selected classification criteria.

1.4.3 Contribution 3: Use Case Maps as a Property Description Language

We propose an abstract high level pattern-based approach to the description of property specifications based on the Use Case Maps. We present a set of commonly used properties with their specifications that are described in terms of occurrence, ordering and temporal scopes of actions. Furthermore, the proposed approach also supports the description of properties with respect to their architectural scope. This contribution is published in: **Use Case Maps as a property specification language (Journal of Software and System Modeling (SoSyM) [HRD07b]**.

1.4.4 Contribution 4: Illustrative Experiments of Early Stages V&V methodology

The Early Stages V&V approach and its supporting techniques have been validated against a different types of applications. Chapter 11 includes results and lessons learned from these experiments:

- IP Multicast Routing Protocol.
- Business process modeling (an online Store).

1.5 Issues not Addressed in this Thesis

The following issues are not addressed in our research:

- The use of UCMs for capturing requirements and eliciting system scenarios.
- Validation of the UCM prototype against the informal functional requirements.
- The integration of the scenarios (for instance repaired slices) into the UCM system specification.

1.6 Thesis Outline

The remaining parts of the thesis are divided into ten chapters:

- Chapter 2 presents general definitions of concepts used throughout the thesis as well as an introduction to Use Case Maps and Abstract State Machines.
- Chapter 3 provides a literature review covering background relevant to the thesis, in particular information on scenario notations, on formal semantics, on formal description techniques, and on validation and verification.
- Chapter 4 presents the formalization of Use Case Maps in terms of Abstract State Machines.
- Chapter 5 describes the proposed validation approach. Techniques and steps are illustrated using the Simple Telephone System example introduced in Chapter 2.
- Chapter 6 presents a literature survey of timed scenario languages. We propose eleven classification criteria to categorize and compare thirteen timed scenario languages.
- Chapter 7 discusses time extension alternatives and how they fit in the context of UCM. Then, the proposed timed version of UCM is formalized using Clocked Transition Systems (CTS) and Timed Automata(TA) formalisms.
- Chapter 8 presents an approach to formally verify properties against timed UCM specifications using model checking technique. We show how change impact analysis combined with reduction techniques can help verify UCM models more efficiently.
- Chapter 9 presents a survey of existing specification patterns and proposed an approach to use UCMs as a property specification language to model requirement properties.
- Chapter 10 presents two experiments used to validate the Early Stages V&V methodology.
- Chapter 11 recalls the contributions of the thesis, compares the Early Stages V&V methodology to similar approaches, and attempts to provide new insights in how to integrate Early Stages V&V methodology to design processes with a wider scope. This chapter concludes with some directions for future research.

Chapter 2

Basic Definitions and Notations

This chapter provides general definitions of concepts used throughout the thesis as well as introductions to Use Case Maps and Abstract State Machines.

2.1 Introduction to Use Case Maps

This section presents Use Case Maps notation as introduced by Buhr and Casselman [BC96], followed by the current state of the art in Use Case Maps semantics. Finally, tools that support Use Case Maps are presented.

2.1.1 Philosophy of UCMs

The Use Case Maps language is a high-level design language that helps humans to express and reason about a system's large-grained behavior patterns. UCMs link high level system behavior and architecture in an explicit and visual way. System functionalities are expressed in terms of causal relationships between responsibilities along scenario paths. The relationships are said to be causal because they involve concurrency and partial ordering of activities and because they link causes (e.g., preconditions and triggering events) to effects (e.g. postconditions and resulting events).

Use Case Maps bridge a modeling gap between requirements (expressed with prose use cases) and design (expressing realization details) [AM01b]. Hence, designers do not have to commit too early on the architecture and the exchange of messages between the different entities. Buhr and Casselman [BC96] claim that details tend to obscure the big picture at stages in the design process where the big picture is the focus of concern. Figure 2.1 [BC96] shows the need for a level between requirements and detailed design in a pyramid of four levels of design abstraction (requirements, high-level design, detailed design and implementation).

Figure 2.2 [MAB$^+$01] represents a simplified call connection to illustrate the use of UCM. This scenario is not necessarily bound to one specific architecture and therefore called *unbound UCM.* Alternative architectures can be developed for the same UCM, for early architectural reasoning. In Figure 2.3a, for instance, the UCM path from Figure 2.2 is bound to two users connected through

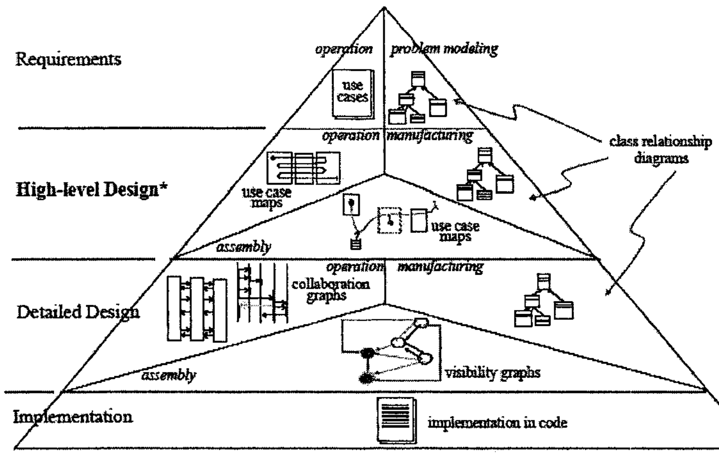


Figure 2.1: A Suite of Design Models [BC96]

an agent-based architecture, whereas Figure 2.3b uses a more conventional architecture based on Intelligent Networks (IN).

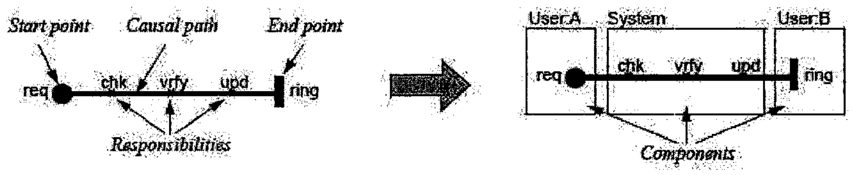


Figure 2.2: A Simple UCM [MAB$^+$01]

These *bound* UCMs may be further refined in models for detailed design (e.g. with Message Sequence Charts (MSC) [IT04]). Figures 2.3c and 2.3d refine respectively the scenarios in Figure 2.3a and Figure 2.3b in terms of MSCs where complex protocols or negotiation mechanisms are used between different system components. Miga et al. [MAB$^+$01] proposed a method to derive Message Sequence Charts [IT04] from Use Case Maps scenario specifications. In a recent work Kealey and Amyot [KA07] have proposed enhanced UCM trace transformations to Message Sequence Charts.

2.1.2 UCM Basic Notation

As shown in Figure 2.2, the basic UCM contains at least the following constructs: start points, responsibilities, end points and components.

- **Start points.** The execution of a scenario path begins at a start point. A start point is represented as a filled circle representing preconditions and/or triggering events (e.g. *req* in Figure 2.2).

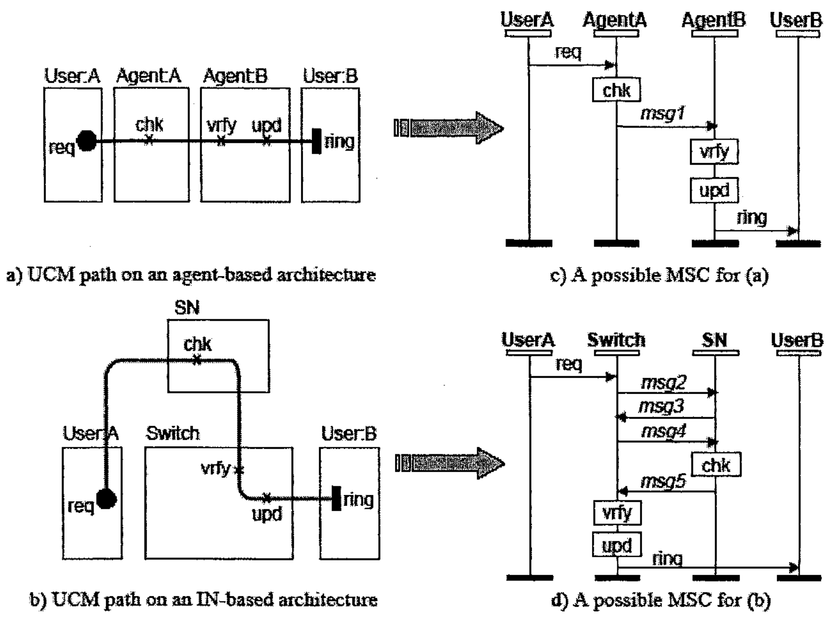


Figure 2.3: UCM Path Bound to Two Different Architectures, and Potential MSCs [MAB+01]

- **Responsibilities.** Responsibilities are abstract activities that can be refined in terms of functions, tasks, procedures, events. Responsibilities are represented as crosses (e.g., *chk* and *upd*).

- **End points.** The execution of a path terminates at an end point. End points are represented as bars indicating post conditions and/or resulting effects(e.g., *ring*).

- **Component**. A UCM component is generic and abstract enough to represent software entities (e.g. object, agent, process, etc.) as well as non software entities (e.g. actors or hardware) (they are represented as simple boxes in Figure 2.2).

UCMs help in structuring and integrating scenarios in various ways— sequentially, as alternatives (with OR-forks/joins as illustrated in Figure 2.4(a)) or concurrently (with AND-forks/joins as illustrated in Figure 2.4(b)).

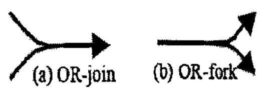
- **OR-Forks.** Represent a path where scenarios split as two or more alternative paths. An OR-Fork has one incoming hyper-edge and two or more outgoing ones. Boolean conditions, called guard, represented as labels between square brackets can be attached to alternative paths.

- **OR-Joins.** Capture the merging of two or more independent scenario paths.

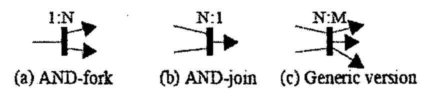
- **AND-Forks.** Split a single control into two or more concurrent scenario paths.

- **AND-Joins.** Capture the synchronization of two or more concurrent scenario paths.

An OR-join merges two (or more) overlapping paths while an OR-fork splits a path into two (or more) alternatives. Alternatives may be guarded by conditions represented as labels between square brackets.



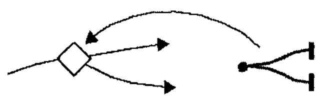
(a) OR-Fork/Joins



(b) Concurrent routes with AND-Fork/Joins

Figure 2.4: Structuring Scenarios

When maps become too complex to be represented as one single UCM diagram, a mechanism for structuring sub-maps becomes necessary. UCM provide the stub concept allowing for hierarchical decomposition of complex maps. UCM path details can be hidden in separate sub-diagrams called *plug-ins*, contained in *stubs* (diamonds) on a path. These plug-ins are reusable UCMs that can be used (plugged) in many stubs. Figure 2.5 illustrates the stub concept.


(a) Static stubs have only one plug-in map

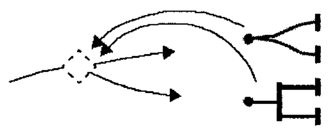

(b) Dynamic stubs may have multiple plug-in maps

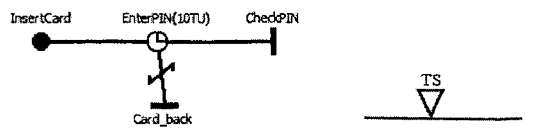
Figure 2.5: Stubs and Plug-in Maps

There are two types of stubs:

- **Static stubs**. represented as plain diamonds (Figure 2.5(a)). They contain only one plug-in map.
- **Dynamic stubs**. represented as dashed diamonds (Figure 2.5(b)). They may contain several plug-in maps, whose selection is determined at run-time according to a selection policy (often described with preconditions).

The Use Case Maps language provides two explicit constructs for expressing time constraints:

- **Timer**: A timer is a waiting place that is triggered by the timely arrival of a specific event. It can also trigger a time-out path when this event does not arrive in time. Figure 2.6(a) illustrates the *Timer* construct, where a timer should start after inserting an ATM card into the bank machine. If the user enters his/her PIN within a 10 Time Units (TU) time frame (*EnterPIN(10TU)*), the PIN will be checked otherwise the card is returned to the user (i.e., time-out path is triggered).



(a) Timer (b) Time Stamp

Figure 2.6: UCM Timed Notation

- **Time Stamp**: A time stamp denotes the time at which a certain event occurred. It can be used to describe response time requirements (see Figure 2.6(b)).

2.1.3 UCM Component Notation

One of the strengths of UCMs resides in their ability to bind responsibilities to components. The default UCM component notation is abstract enough to represent dependencies (for instance containment), different types (passive, active, etc.), and it even allows to represent run-time instances (without data). Components can be of different types and possess different attributes. Buhr in [Buh98] suggests several types and attributes that are relevant for complex systems (real-time, object-oriented, dynamic, agent-based, etc.).

Figure 2.7 illustrates some of these component types and attributes proposed by Buhr [Buh98]:

- *Teams* (boxes with sharp corners) are the most generic component that are also most typically used in UCMs. Teams are operational groupings of system-level components.
- *Objects* (boxes with rounded corners) are data or procedure abstractions that are system-level components to support the system comprehension.
- *Processes* (Parallelograms) are active components.
- *Slots* (boxes with dashed outlines) may be populated with different instances of components at different times. Slots are containers for *dynamic components* (DC) in execution.
- *Pools* are containers that hold components in readiness to occupy slots (e.g., not executing DC, they act as data).
- *Dynamic components* (see Figure 2.7(c)) can be created, moved, stored, and deleted with dynamic responsibilities such as create, put, get, and move. *Move arrows* (small arrows between paths and pools or slots) are used to indicate the possibility of component movement that may cause slots to become occupied or empty. Movement is a metaphor for changing visibility. Moving a component into a slot allows to make this component visible to those who must interact with it at the slot location level.

The slot notation does not indicate whether slots are empty or not. This requires an analysis of the corresponding paths. Therefore, slots can be seen as places where different components may play the same role at different times.

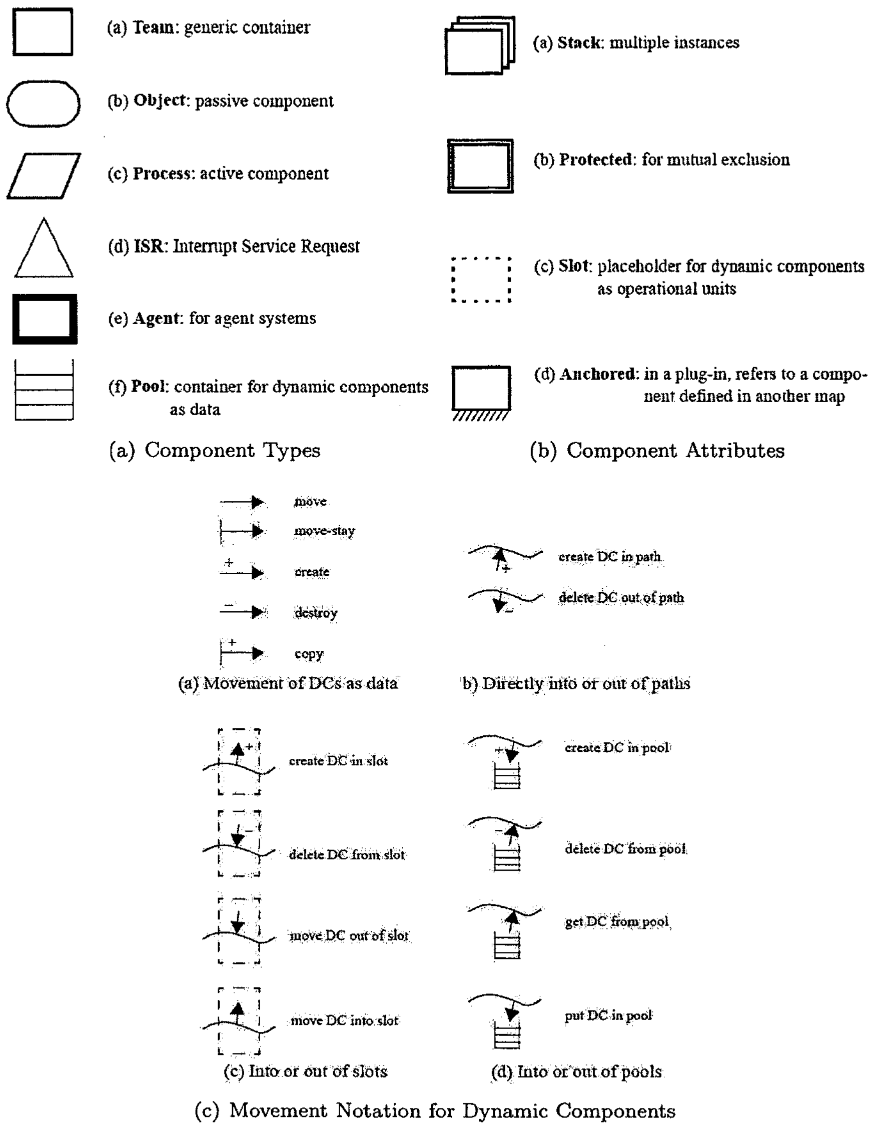
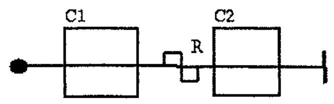

(a) Team: generic container
(b) Object: passive component
(c) Process: active component
(d) ISR: Interrupt Service Request
(e) Agent: for agent systems
(f) Pool: container for dynamic components as data
(a) Component Types
(a) Stack: multiple instances
(b) Protected: for mutual exclusion
(c) Slot: placeholder for dynamic components as operational units
(d) Anchored: in a plug-in, refers to a component defined in another map
(b) Component Attributes
move
move-stay
create
destroy
copy
(a) Movement of DCs as data
create DC in path
delete DC out of path
b) Directly into or out of paths
create DC in slot
delete DC from slot
move DC out of slot
move DC into slot
(c) Into or out of slots
create DC in pool
delete DC from pool
get DC from pool
put DC in pool
(d) Into or out of pools
(c) Movement Notation for Dynamic Components

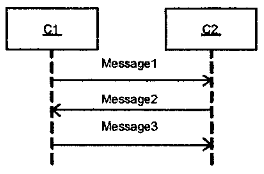
Figure 2.7: Component Types, Attributes and Movement Notation [Amy01b]

Use Case Maps is not an Architecture Description Language (ADL), but a high level visual specification language that helps stakeholders to document and reason about a system-wide architecture and behavior. ADLs represent a formal way of representing architecture with a primary mission of describing components and their connectivity. ADLs permit analysis of architectures completeness, consistency, ambiguity, performance and support automatic generation of software systems. Use Case Maps focus on the behavior of the whole system rather than on their parts.

UCM component relationships depend on scenarios to provide the semantic information about their dependencies. Components are dependent if they share the same scenario execution path. To illustrate the fact that a responsibility is the result of a collaboration among two components, the *shared responsibility* construct is used (see Figure 2.8(a)). The execution of a shared responsibility requires message-like interactions between the involved components. Figure 2.8(b) shows one possible refinement of the shared responsibility in terms of a sequence diagram.



(a) Shared Responsibility R



(b) Refinement of R in terms of sequence diagram

Figure 2.8: Shared Responsibility and One Possible Refinement

Note: Communication links (e.g., physical links) between components are usually not required, but they can be added.

2.1.4 Running Example: A simple Telephone System

This section illustrate some of the basic UCM concepts using a UCM model (originally introduced in [MAB+01]) describing the connection request phase in an agent based telephony system with user-subscribed features. This UCM model will be used as an ongoing example throughout the thesis.

It contains four components (originating/terminating users and their agents) and two static stubs. Upon the request of an originating user (req), the originating agent will select the appropriate user feature (in stub Sorig) that could result in some feedback. This may also cause the terminating agent to select another feature (in stub Sterm) which in turn can cause a number of results in the originating and terminating users. Stub *Sorig* contains the originating plug-in map whereas stub *Sterm* contains the Terminating plug-in map. These sub-UCMs have their own stubs, whose plug-in maps are user-subscribed features.

1. Stub *Sscreen*:

    • OCS (Originating Call Screening): blocks calls to people on the OCS filtering list.

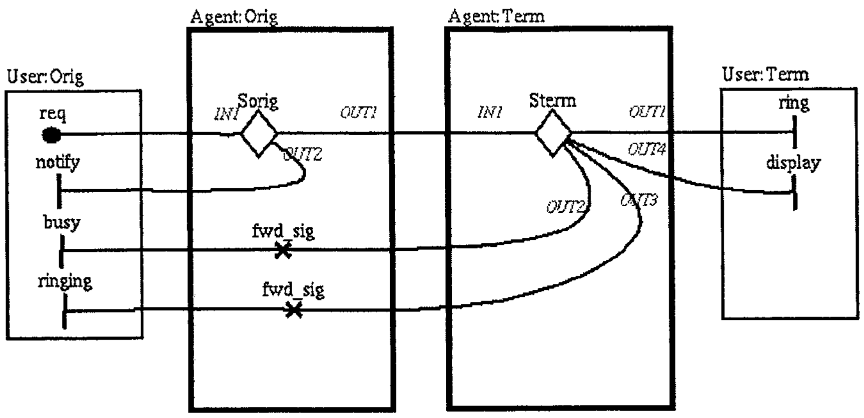


Figure 2.9: Simple Telephone System Root Map

- Default: used when not subscribed to any other originating feature.

2. Stub *Sdisplay*:

    - CND (Call Name Delivery): displays the caller's number on the callee's device (display) concurrently with the rest of the scenario (ringing).
    - Default: used when not subscribed to any other terminating feature.

The set of global variables for the UCM map are: **Busy** (the callee is busy), **InOCSList** (the callee is on OCS list), **subCND** (the callee is subscribed to CND), **subOCS** (the caller is subscribed to OCS).

Note: Use Case Maps does not have a notion of local variables.

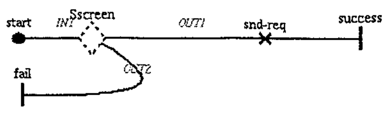
Each plug-in map (Fig. 2.10) is bound to its parent stub, i.e. stub input/output segments (IN1, OUT1, etc.) are connected to the plug-in map start/end points, as follows:

- Sorig Stub: Originating UCM. Condition: *true.*
  Binding:((IN1, start), (OUT1, success), (OUT2, fail))

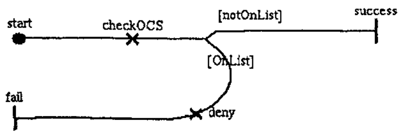
    - Sscreen Stub:
        * OCS UCM: Condition: subOCS. Binding: ((IN1, start), (OUT1, success), (OUT2, fail))
        * Default UCM: Condition: ¬ subOCS. Binding: ((IN1, start), (OUT1, continue))

- Sterm Stub: Terminating UCM. Condition: True. Binding: ((IN1, start), (OUT1, success), (OUT2, fail), (OUT3, reportSuccess), (OUT2, disp))

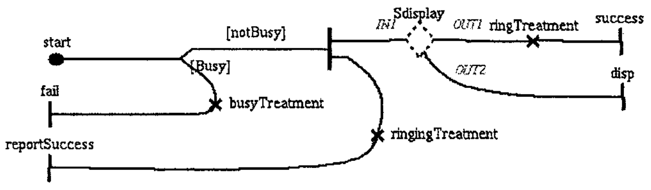
    - Sdisplay Stub:
        * CND UCM: Condition: subCND. Binding: ((IN1, start), (OUT1, success), (OUT2, disp))
        * Default UCM: Condition: ¬ subCND. Binding:((IN1, start), (OUT1, continue))



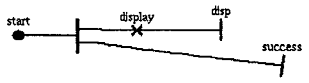
(a) Originating plug-in map



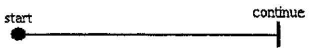
(b) OCS plug-in map



(c) Terminating plug-in map



(d) CND plug-in map



(e) Default plug-in map

Figure 2.10: Simple Telephone System Plug-in Maps

2.1.5 Use Case Maps Properties

The main properties and strengths of UCMs can be summarized as follows:

- Scenarios are represented as architectural entities that combine both behavior (set of paths) and structures (UCM components) into a big picture. This promotes system comprehension without having to mentally integrate information from different diagrams [Buh98].
- UCMs provide an integrated view of scenarios. This makes the notation more abstract and more useful with respect to architectural considerations [BC96].
- UCMs provide a visual and explicit way to represent causality. This reduces the mental effort required to draw the big picture. Within one component a causal path is viewed as a state transitions, while between components a causal path is viewed as a messages exchange [Buh98].
- UCMs abstract the system behavior and architecture from details. System behavior is expressed above the level of inter-component message exchange (i.e., communication protocols), objects creation and deletion, communication constraints, data and control. While in [BC96] a number of different universal component types is provided, system architecture can still be expressed in terms of rectangular boxes representing any type of runtime component implemented in either software or hardware.
- UCMs may provide helpful visual patterns that stimulate thinking and discussion about system issues and that may be reused [Buh98].
- UCM dynamic stubs help specify how alternative scenarios could evolve at run time. It promotes the early thinking (i.e., at design time) about potential conflicts that could arise during

the system dynamic execution. The selection policies can help avoid or resolve these conflicts [AM01b].

- UCMs may be usefully integrated with other high-level techniques such as Message Sequence Charts [MAB$^+$01, KA07].

2.1.6 Use Case Maps Semantics

UCM abstract syntax and static semantics are informally defined in the draft standard (Z.152) [IT02b]. This draft includes an XML Document Type Definition (DTD) for UCM. This DTD [AM01a] proposes an XML-based interchange format for UCM tools. However, implementation details are absent from this document. Another XML DTD [AHHC03] was proposed to describe the export format of scenarios resulting from a UCM traversal. Based on the Z.152 draft and the UCM scenario DTDs, a UCM meta-model was proposed by Zeng [Zen05] and Bo [Jia05] as a result of a joint collaboration (depicted in Figure 2.11).

Figure 2.12 illustrates a more recent UCM meta-model that is part of URN meta-model [jUC07]. *PathNode* represents the parent class for all UCM constructs. For UCM performance annotated meta-models, the reader is invited to consult [Jia05].

Dynamic semantics in UCM are still informal, although it has been indirectly introduced in terms of the formal language LOTOS [ISO89]. [ABBL95] and [Amy94] formalized UCMs using LOTOS. UCMs have been baptized URN-FR, while another and complementary notation for non-functional requirements [CNYM99] (GRL – Goal-oriented Requirements Language [IT03b]) is called URN-NFR. URN-NFR is out of the scope of this thesis.

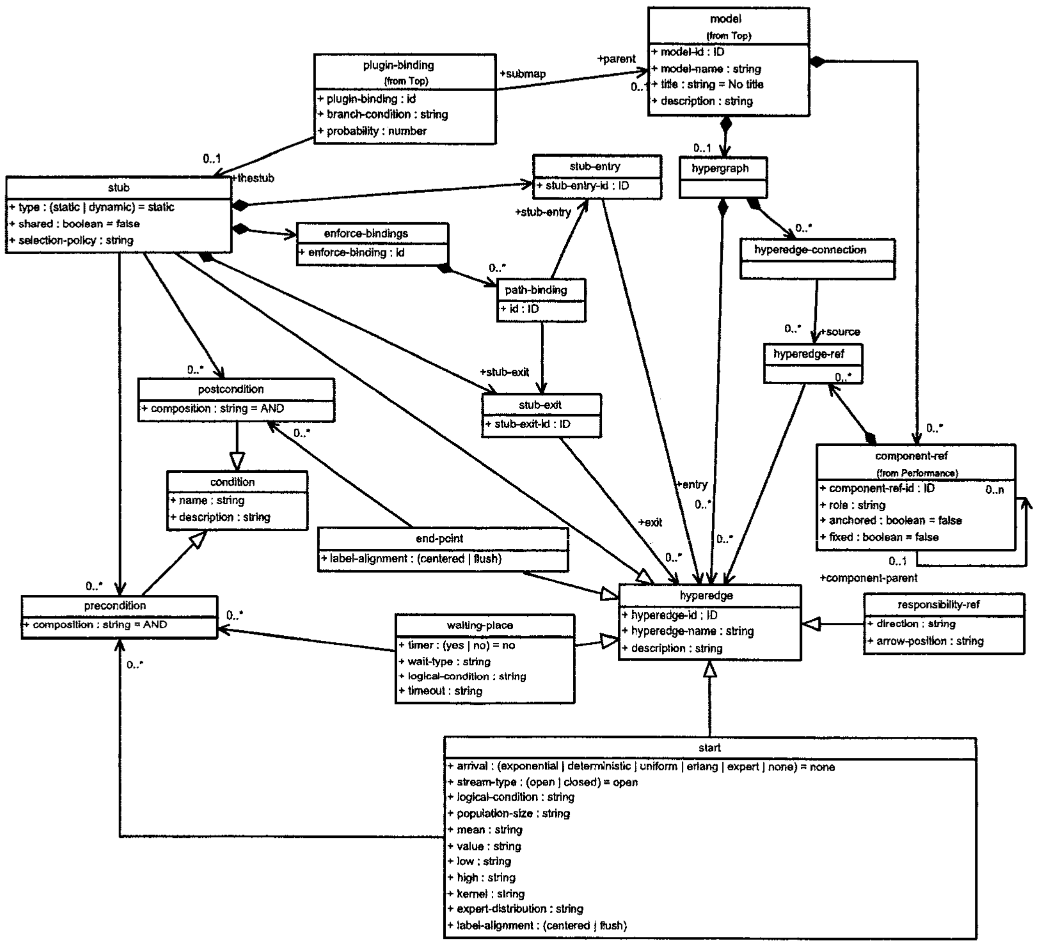
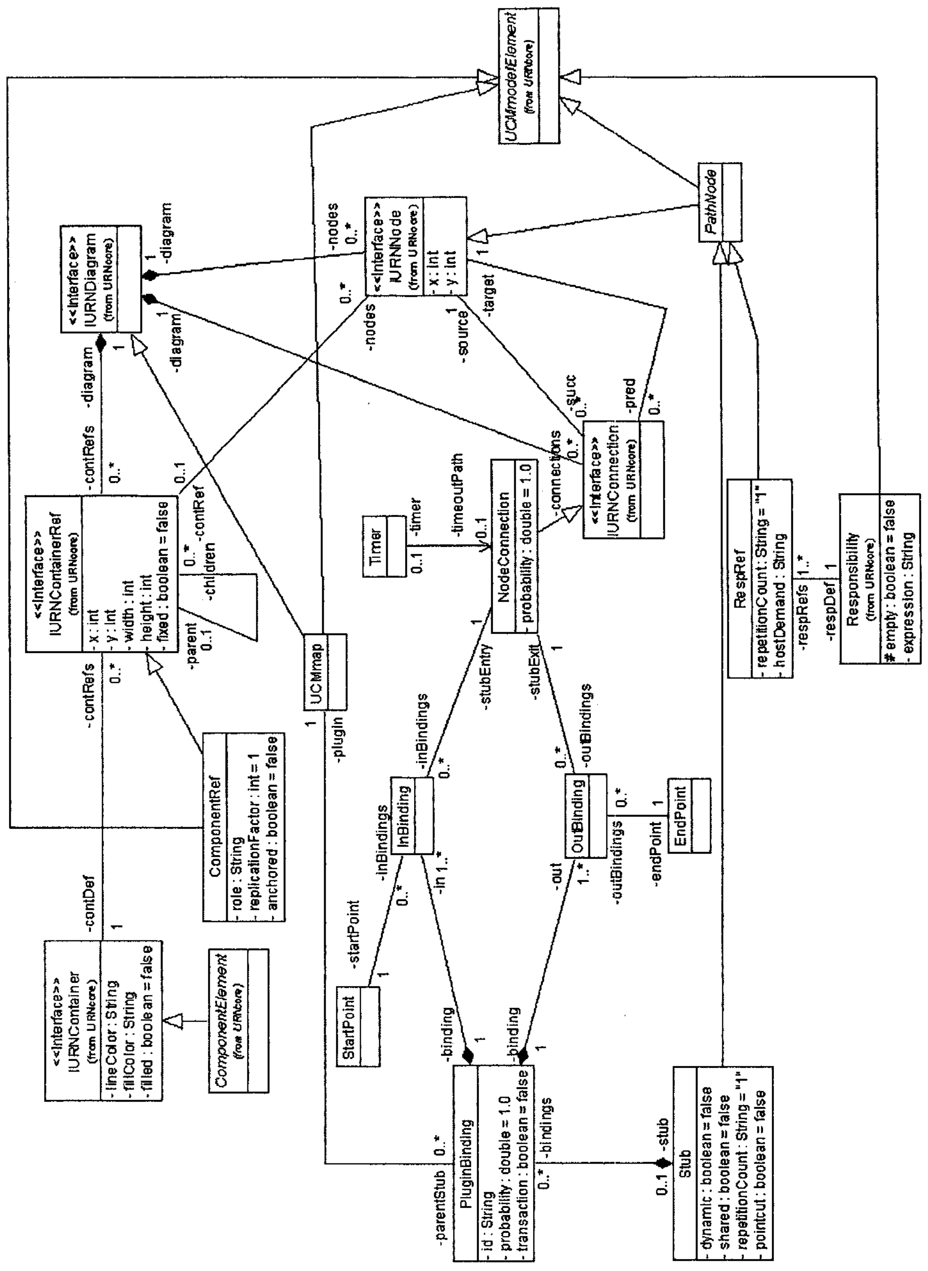


Figure 2.11: UCM Path Meta-Model [Zen05, Jia05]



UCMmodelElement (from URNcore)
PathNode
<<Interface>> IURNDiagram (from URNcore)
<<Interface>> IURNNode (from URNcore)
- x : int
- y : int
<<Interface>> IURNConnection (from URNcore)
<<Interface>> IURNContainerRef (from URNcore)
- x : int
- y : int
- width : int
- height : int
- fixed : boolean = false
ComponentRef
- role : String
- replicationFactor : int = 1
- anchored : boolean = false
<<Interface>> IURNContainer (from URNcore)
- lineColor : String
- fillColor : String
- filled : boolean = false
ComponentElement (from URNcore)
UCMmap
Timer
NodeConnection
- probability : double = 1.0
RespRef
- repetitionCount : String = "1"
- hostDemand : String
Responsibility (from URNcore)
# empty : boolean = false
- expression : String
InBinding
OutBinding
EndPoint
StartPoint
PluginBinding
- id : String
- probability : double = 1.0
- transaction : boolean = false
Stub
- dynamic : boolean = false
- shared : boolean = false
- repetitionCount : String = "1"
- pointcut : boolean = false
-diagram
-nodes
-contRefs
-contRef
-children
-parent
-contDef
-source
-target
-succ
-pred
-connections
-timer
-timeoutPath
-stubEntry
-stubExit
-inBindings
-outBindings
-in
-out
-startPoint
-endPoint
-binding
-bindings
-parentStub
-plugin
-stub
-respRefs
-respDef

Figure 2.12: UCM Path Meta-Model [jUC07]

2.1.7 UCM Tools

The UCM language is supported by a freely available editing tools: UCMNav (UCM Navigator) [Mig98] and jUCMNav [jUC06]. In this thesis, both tools were used to create and maintain UCM models.

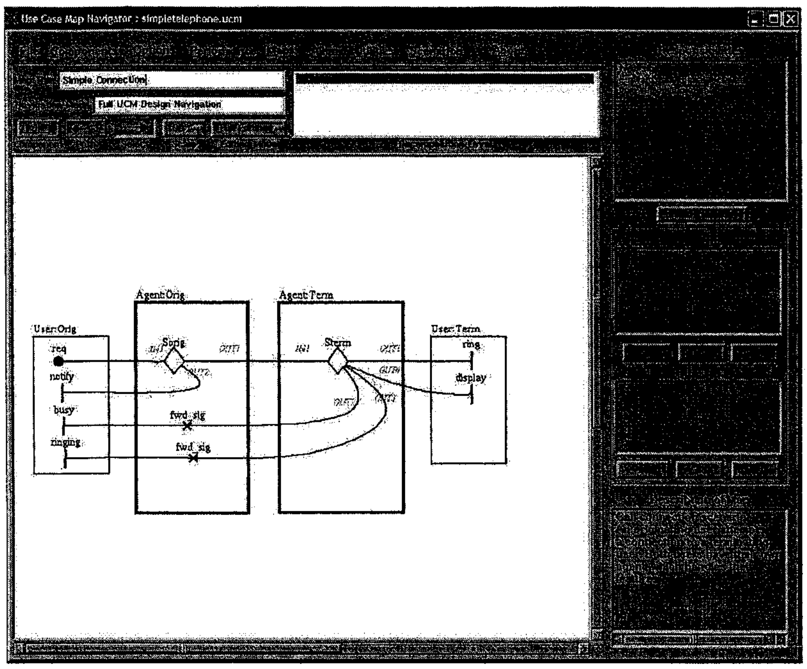


Figure 2.13: UCMNav GUI

UCM Navigator

UCMNav [Mig98], developed by *Andrew Miga* at *Carleton University*, supports the creation, navigation, and maintenance of UCMs. The following list summarizes some of the functionalities of UCMNav:

- Both the path and component notations are fully supported. It gives the possibility to add comments and descriptions of the design and/or individual elements.
- It ensures the syntactical correctness of the UCMs manipulated with respect to DTD [AM01a].
- It maintains various kinds of bindings (plug-in maps to stubs, responsibilities to components, sub-components to components, etc.).
- It allows users to visit and edit the plug-in maps related to stubs at all levels.
- It generates XML descriptions and exports UCMs in different formats (e.g. Encapsulated Postcript(EPS), Computer Graphics Meta-file(CGM), Scalable Vector Graphics(SVG), and Maker Interchange Format(MIF)).

- It supports a simple data model and scenario definitions. This feature enables the highlight of specific scenario paths in a UCM specification.

- It allows the generation of refined models in terms of MSCs [MAB+01].

In recent years, there has been many additions to *UCMNav*. For instance, Petriu [PW02] added an *UCM2LQN* export mechanism to *UCMNav*. *UCM2LQN* is an automated conversion tool that converts UCM performance annotated models into LQN performance models. It works as a link between the *UCMNav* and two LQN analysis tools: *LQNS* and *ParaSRVN*. Jiang [PAWJ03] recently added an export mechanism to integrate UCM models with other requirements in a requirements management system (Telelogic DOORS [AB02]). Echihabi [DAH04] developed a complementary tool called *UCMExporter* to transform UCM scenarios into TTCN [IT03a] and MSC [MAB+01]. Multiple platforms are currently supported by *UCMNav*: Solaris, Linux (Intel and Sparc), HP/UX, and Windows (95, 98, 2000, XP and NT). Figure 2.13 shows *UCMNav* graphical user interface.

jUCMNav

jUCMNav [jUC06] is an open-source tool for editing and analyzing *URN* models. This tool is a plug-in developed with and for the Eclipse framework [IBM06a, IBM06b], an extensible Java-based development platform. *jUCMNav* [RKA06] was first developed to support the creation and maintenance of Use Case Maps scenario models, but GRL [IT03b] was recently added to achieve complete coverage of URN.

The integration of UCM and GRL views in the same tool allows for the creation of various types of traceability links between elements of both notations. These links can be used to measure the impact of a modification to any evolving GRL/UCM diagram on the other aspects of the model. For instance, links can be defined between GRL intentional elements or actors as source, and UCM responsibilities, components, or maps as target. Very recently, *jUCMNav* was extended to support scenario definitions [KA07]. Figure 2.14 shows *jUCMNav* graphical user interface.

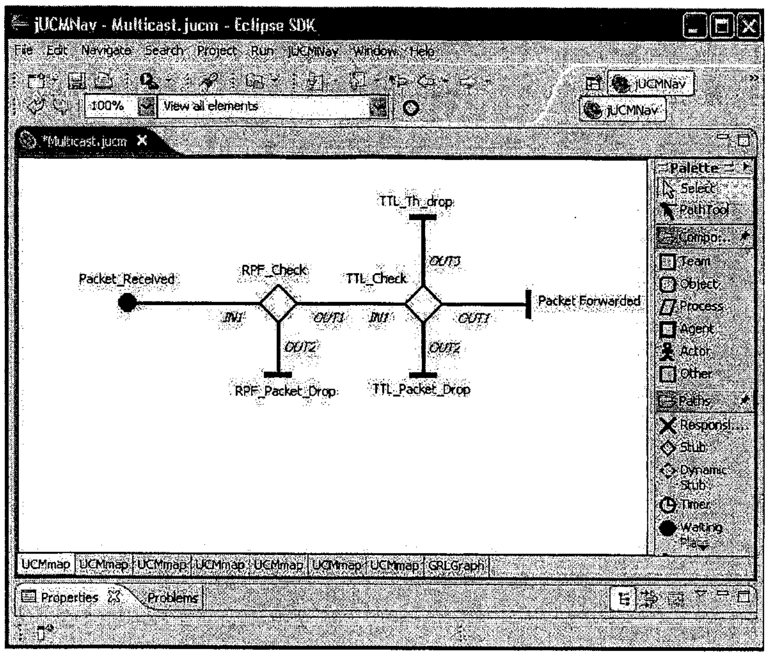


Figure 2.14: jUCMNav GUI

2.2 Introduction to Abstract State Machines

Abstract State Machines (ASM) aim to bridge the gap between informal and formal descriptions. ASMs have a precise semantics in order to prevent ambiguities and lead to understandable models. They are expressive enough for modeling various problems and can describe large systems through structuring mechanisms. ASMs provide abstraction and refinement techniques to support modeling at different abstraction levels.

2.2.1 ASM Thesis

Abstract State Machines (ASMs), formerly known as *Evolving Algebras*, have been discovered by Yuri Gurevich [Gur88] in an attempt to improve on Turing's thesis [Tur36] so that:

Every algorithm is an ASM as far as the behavior is concerned. In particular the given algorithm can be step-for-step simulated by an appropriate ASM [Gur04]. (This is the ASM Thesis)

This means that an activity which is conceptually done in one step can be executed in the model in one step. This is in contrast to Turing machines where simple operations might need any finite number of steps.

Abstract State machines have been used to capture sequential, parallel and distributed algorithms. The definition of sequential ASMs was formulated in [Gur94, Gur99, Gur00]. It stipulates that for every sequential algorithm, there exists an equivalent sequential ASM (i.e. with the same set of states, the same set of initial states, and the same state transformation rules). This Sequential ASM Thesis relies upon the following three postulates for sequential algorithms from which it can be proved:

- **Sequential Time Postulate [Gur00].** The sequential time postulate expresses that the behavior of a sequential time algorithm is determined by the set of states, the subset of initial states, and the state transition function.

  Postulate 1 (Sequential Time Postulate) *An algorithm $\mathcal{A}$ is determined by:*

  - *A set $\mathcal{S}(\mathcal{A})$ of states,*
  - *A subset $\mathcal{I}(\mathcal{A})$ of states, called the initial states of $\mathcal{A}$,*
  - *A function τ_A:$\mathcal{S}(\mathcal{A}) \Rightarrow \mathcal{S}(\mathcal{A})$ called the one-step transformation of $\mathcal{A}$.*

- **Abstract State Postulate [Gur00].** The abstract-state postulate requires that the states of a sequential algorithm are first-order structures, with fixed domain and signature, and closed under isomorphisms (respecting the initial states and the state transformation law).

  Postulate 2 (Abstract State Postulate) *Let $\mathcal{A}$ be an algorithm.*

  - *States of $\mathcal{A}$ are first-order structures.*
  - *All states of $\mathcal{A}$ have the same vocabulary.*
  - *The one-step transformation τ_A does not change the base set of any state.*
  - *$\mathcal{S}(\mathcal{A})$ and $\mathcal{I}(\mathcal{A})$ are closed under isomorphisms. Further, any isomorphism from a state X onto a state Y is also an isomorphism from $\tau_A(X)$ onto $\tau_A(Y)$.*

- **Bounded Exploration Postulate [Gur00].** The bounded exploration postulate states that for every sequential algorithm, the transformation law depends only upon a finite set of terms over the signature of the algorithm, in the sense that there exists a finite set of terms such that for arbitrary states X and Y which assign the same values to each of these terms, the transformation law triggers the same state changes for X and Y.

  Postulate 3 (Bounded Exploration Postulate) *Let $\mathcal{A}$ be an algorithm. There exists a finite set T of terms in the vocabulary of $\mathcal{A}$ such that $\Delta(A,X) = \Delta(A,Y)$ whenever states X, Y of $\mathcal{A}$ coincide over T. Δ is a set of updates.*

- **Parallel Characterization Thesis [BG03].** The definitions of parallel ASMs and distributed ASMs were formulated in [Gur94, BG03]. The authors in [BG03] define a parallel algorithm as anything satisfying the sequential time postulate, the abstract state postulate, and several other postulates describing how the parallel subprocesses communicate with each other. The definition of parallel ASMs in [BG03] is a variant of that in[Gur94]. In either version, parallel ASMs are parallel algorithms.

Postulate 4 (Parallel Characterization Thesis) *[BG03] For every parallel algorithm, there is a behaviorally identical parallel ASM.*

The axioms provided in [BG03] do not allow a parallel algorithm to create components on the fly. In [BG07] the authors removed this restriction by liberalizing the axioms provided in [BG03]. Both sequential and parallel ASM thesis have been confirmed theoretically [Gur00, BG03] and Experimentally [Hug06, BS03]. For a rigorous mathematical definition of the semantic foundations of ASMs, we however refer to [BS03, Gur94, Gur00, BG03].

ASMs have been used to specify semantics of a wide variety of programming languages in particular C++ [Wal95] and Java [BS98], logic programming languages such as Prolog [BR95] and its variants, and hardware languages such as VHDL [GBM95]. ASMs have been also used to define the operational semantics of UML activity diagrams [BCR00a] and the formal definition of ITU−T standard SDL 2000 [GK97, EGG+01].

Part of the ASM definition given in this section, was the one-step tranformation τ_A. There was no restriction on how τ should be defined as long as this was done unambigously. In the following section, the semantics of tranformation τ will be given in the context of an abstract programming language.

2.2.2 ASM Program

An ASM define a state-based computational model, where computations (runs) are finite or infinite sequences of states $\{S_i\}$ obtained from a given initial state S_0 by repeatedly executing transitions δ_i.

$$S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \qquad \cdots \qquad \xrightarrow{\delta_n} S_n$$

An ASM $\mathcal{A}$ is defined over a fixed vocabulary V, a finite collection of function names and relation names. Each function symbol has a fixed arity n and type $T_1,\ldots,T_n \rightarrow$ T where T_i and T are basic types. Functions in V may be:

- **Static:** having the same (fixed) interpretation in each computation state of A,
- **Dynamic:** where function names can be altered by transitions fired in a computation step. Dynamic functions can be further classified into:
  - Input functions functions that $\mathcal{A}$ can only read, which means that these functions are determined entirely by the environment of $\mathcal{A}$. They are also called monitored.
  - Controlled functions of $\mathcal{A}$ are those which are updated by some of the rules of $\mathcal{A}$ and are never changed by the environment.
  - Output functions of $\mathcal{A}$ are functions which $\mathcal{A}$ can only update but not read, whereas the environment can read them (without updating them).
  - Shared functions are functions which can be read and updated by both $\mathcal{A}$ and the environment.

Given a vocabulary, $\mathcal{A}$ is defined by its program $\mathcal{P}$ and a set of distinguished initial states S_0. The program $\mathcal{P}$ consists of transition rules and specifies possible state transitions of $\mathcal{A}$ in terms of finite sets of local function *updates* on a given global state. Such transitions are atomic actions. A transition rule that describes the modification of the functions from one state to the next has the following form:

$$\textbf{if } \ Condition \ \textbf{ then } \ <Updates> \ \textbf{ else } \ <Updates> \ \textbf{ endif} \tag{1}$$

Where *Updates* is a set of function updates (containing only variable free terms) of form: $f(t_1,t_2,\ldots,t_n){:=}\ t$ which are simultaneously executed when *Condition* (called also *guard*) is true. In a given state, first all parameters t_i, t are evaluated to their values, v_i, v, then the value of $f(v_1,\ldots,v_n)$ is updated to v. Such pairs of a function name *f*, which is fixed by the signature, and an optional argument $(v_1,\ldots,v_n)$, which is formed by a list of dynamic parameters value v_i, are called *locations*.

Example: The following rules yield the update-set {(x, 2), (y(0), 1)}, if the current state of the ASM is {(x, 1), (y(0), 2)}:

$$\begin{aligned} \textbf{if} \quad (x = 1) \quad \textbf{then} \quad & x := y(0) \\ & y(0) := x \end{aligned} \tag{2}$$

In every state, all the rules which are applicable are simultaneously applied (if the updates are consistent) in one step to produce the next state. Each function update changes a value at a specific location given by the left-hand-side of the update.

A set of updates is called *consistent* if it contains no pair of updates with the same locations, i.e. no two elements (loc,v),(loc,v') with v≠v'. In the case of inconsistency, the computation does not yield a next state.

Example: The following rules yield the inconsistent update-set {(x, 1), (y, 3), (x, 2)}, due to the conflicting updates for x:

$$\begin{aligned} x &:= 1 \\ y &:= 3 \\ x &:= 2 \end{aligned} \tag{3}$$

ASM Universe. ASMs are multi-sorted based on the notion of universes (or domains). Functions are defined over these universes. We presume the standard mathematics universes of Booleans, integers, lists, etc., as well as the standard operations on them such as the usual Boolean operations ($\wedge$, $\vee$, etc.). A universe can be dynamically extended with individual objects by:

$$\begin{aligned} &\textbf{extend}\ \ \textit{Universe}\ \ \textbf{with}\ \ v \\ &\qquad < \textit{Rule} > \\ &\textbf{end extend} \end{aligned} \tag{4}$$

where v is a variable which is bound by the extend constructor.

Synchronous Execution. Simultaneous updates provide a convenient way to abstract from sequentiality. This feature allows for the possible description of parallel and distributed systems. It is enhanced by the ASM construct *forall* allowing the simultaneous execution of a rule R for each element v satisfying a given condition ϕ.

$$\begin{aligned} &\textbf{forall}\ \ x\ \ \textbf{with}\ \ \phi \\ &\qquad \textit{Rule} \end{aligned} \tag{5}$$

Non-determinism in ASMs. A convenient way to abstract from details of scheduling of rule executions is to use non-determinism. In ASM, non-determinism can be introduced in two ways: by using input function (external functions), which serve as oracles in that their value is determined through the environment, and by using the *choose* constructor which defines an arbitrary selection of one element in a universe:

$$\begin{aligned} &\textbf{choose}\ \ v\ \ \textbf{in}\ \ \textbf{Universe}\ \ \textbf{with}\ \ \phi \\ &\qquad \textit{Rule} \end{aligned} \tag{6}$$

where v is non-deterministically selected from the given universe. The *choose* constructor can be qualified by a condition.

2.2.3 Multi-Agent ASMs

Multi-Agent Abstract State Machines, also called distributed ASMs, allow for multiple concurrent sequential computations of single agents, each executing its own sequential ASM.

For Multi-Agent ASMs, the notion of run, which is defined for sequential ASMs as sequence of computation steps of a single agent, is replaced by the notion of a *partial order* of moves of finitely many agents.

Two types of multi-agents are defined in the literature [BS03]:

- Synchronous Multi-Agent ASMs: Defined as a set of agents running in parallel synchronized using an implicit global system clock. The sequence of events determining a run is the sequence of states forming the run of the underlying multi-agent synchronous ASM, where the global clock plays the role of a step counter.

- Asynchronous Multi-Agent ASMs: Defined as agents proceeding in parallel at their own pace and with atomic actions applied in its own local states, including input from the environment as monitored functions. More formally an asynchronous ASM is given by a family of pairs (a, ASM(a)) of pairwise different agents, elements of a possibly dynamic finite set AGENT, each executing its basic program ASM(a)

2.2.4 ASM Properties

The main properties and strengths of ASMs are summarized in the following points:

- Using ASMs, we can express a system structure as well as its dynamics within the same notational framework.
- ASM are easy to understand and can be used by engineers that are not familiar with algebra and logic since the overall framework is kept much simpler.
- ASM can be seen as a general purpose notation. It is suitable for various kinds of systems in terms of the problem to be solved and also in terms of size (the broad range in the literature shows this, see [Hug06]).
- Model on a natural level of abstraction. Details may be abstracted in the model for the benefit of conciseness and readability.
- ASMs provide a proof of correctness of the system model in several steps. The system is described at different levels of abstraction, where each level is an extension or refinement of the next upper level by means of an additional feature that is specified. By proving the correctness between models on each two adjacent levels of abstraction, we obtain the global proof.

2.2.5 ASM Tools

Several ASM engines were designed to develop and validate Abstract State Machines specifications. The main known ones are:

- ASM Workbench (ASM-WB) [Cas99] designed by Giuseppe Del Castillo at the University of Paderborn (Germany).
- ASM Gopher [Sch06] designed by Joachim Schmid and Wolfram Schulte at the University of Ulm, (Germany). It is an extension of the functional programming language Gofer.
- XASM (eXtensible Abstract State Machines) [Anl00] designed by Matthias Anlauff at the Technical University of Berlin (Germany). XASM became an open-source ASM tool.
- AsmL (Abstract State Machines Language) [ASM06], developed by the Foundation of Software Engineering group at Microsoft Research.

More information about these and other ASM tools can be found in [Hug06]. In the following section, we provide a brief introduction to AsmL [ASM06], which is utilized in this thesis.

AsmL: The Abstract State Machine Language

AsmL is an executable specification language based on the theory of Abstract State Machines. The current version, AsmL for Microsoft .NET, is embedded into Microsoft Word. It uses XML and Word for literate specifications. It is fully interoperable with other .NET languages. AsmL generates .NET assemblies which can either be executed from the command line, linked with other .NET assemblies, or packaged as COM components. AsmL is integrated with *Spec Explorer* [Spe06], a software development tool for advanced model-based specification and conformance testing. Spec Explorer is the successor of AsmL for Microsoft .NET. It contains a compiler for AsmL, and a tool to explore models written in AsmL.

AsmL has a strong mathematical component. It is fully object-oriented as well, and it provides complex data types such as sets, finite mappings, sequences and structures.

The crucial features of AsmL, intrinsic to ASMs, are massive synchronous parallelism and finite choice. ASMs steps are transactions, and in that sense AsmL programming is transaction programming.

Here are some additional features of AsmL:

- Advanced type system: disjunctive types, semantic subtypes, generics,
- Pattern matching for structures and classes,
- Intra-step communication with outside world and among sub-machines,
- Reflection over execution,
- Data access, structural coverage,
- States as first class citizens.

2.3 Chapter Summary

This chapter introduced material for readers who want to familiarize themselves with *UCMs* and *Abstract State Machines.* It covers the philosophy behind each notation, the information needed to use them, elements of the notation (paths and components for UCMs; universes and rules for ASM), and tool support.

Chapter 3

State of the Art

This chapter provides a literature survey of state of the art methodologies that are closely related to our *Early Stages V&V approach.* First, we provide an overview of several high level scenario based notations, discuss scenario integration approaches and present existing classification approaches. Then we provide an overview of formal semantics approaches as well of some formal specification techniques. Finally, we present the SPEC-VALUE approach proposed by Daniel Amyot in his PhD thesis [Amy01a].

3.1 Scenario Notations

Scenarios are known to help requirements engineers to elicit functional requirements, uncovering hidden requirements and trade-offs, as well as comprehend and validate requirements. Scenario-based models are intuitive to use and improve the communication of requirements to stakeholders. They go beyond the requirement phase by covering the whole software development life cycle. They drive the specification, design, testing, validation, and the evolution of systems. The exact definition of a scenario may vary depending on purpose, contents and used semantics [RP96], but most definitions include the notion of a partial description of system usage as seen by its stakeholders.

With the advent of Object-oriented design modeling more than a decade ago, the concept of *use cases* [JBR99] become a widespread practice for capturing functional requirements. Most authors agree that, in broad terms, use cases and scenarios are descriptions of a sequence of actions or events of some generic task which the system is meant to accomplish. However, there is no agreed distinction between the meanings of use case and scenario.

Jacobson et al. [JEJ94], define use cases as follows: "a use case is a sequence of transactions in a system whose task is to yield a measurable value to an individual actor of the system.". In UML context, Rumbaugh et al. [RJB99] define a scenario as a sequence of actions that illustrates behavior. A scenario may be used to illustrate an interaction or the execution of a use case instance [RJB99]. A slightly different distinction between "use cases" and "scenarios" is stated by Maiden et al. [Mai98]. They treat use cases as a collection of actions and the temporal rules that govern how the actions can be linked together. In contrast, a scenario is one sequence of events, the ordering of which is tied to the start and the end events or actions in the use case.

Note: In this thesis, the terms *use cases* and *scenarios* are used interchangeably.

In the following section, we give a short overview of some high-level scenario-based notations (e.g., MSC, UML sequence Diagrams, etc.) as well as a literature review of their main formalization attempts.

- **Message Sequence Charts (MSCs)** Message sequence charts (MSCs), standardized by the ITU-T [IT04] in recommendation Z.120, is a popular language for specifying scenarios that describe interactions between system entities. MSCs have been particularly useful in the early stages of system development in domains such as telecommunication. Message Sequence Charts may be used for requirement specification, simulation and validation, test-case specification and documentation of real-time systems.

  Basic MSCs (bMSCs) model the communication behavior of system components (vertical lines) and their environment through message message exchanges (arrows). An MSC is completely characterized by the sequences of events it allows. An event may be a local action, message events (i.e. sending a message, receiving a message, a lost message and a found message), creation and termination of an instance and timer events (i.e. set, reset and time-out) [Mau96].

  A set of bMSCs usually covers a partial system behavior only. However, they can be combined to form more complete specifications by means of High-level Message Sequence Charts (HMSCs) [MR97]. HMSCs enable the structuring and hierarchical decomposition of basic MSCs through alternative, sequential and parallel composition, a family of loop operators, and operators for describing optional behavior and exceptions. MSCs with structures is another MSC variant that contains some structures, such as coregions, references, inline expressions. It mixes events and bMSCs.

  Several approaches to formalizing MSC have been studied. For basic MSCs, semantics have been given in terms of Petri Nets [Hey00, GRG93], Abstract Execution Machine [JP01], automata [LL93a], partial order [AHP96]. The most extensive semantics are based on Process algebra [MR94, Mau96, MR99]. Katoen et al. [KL98] provide a compositional denotational semantics for bMSCs and HMSCs. A bMSC is mapped onto a partially ordered multiset (pomset). The constructors of HMSC correspond to the appropriate operations on pomsets. S. Heymer [Hey98] use labeled partially ordered sets (lposets) to define the semantics of bMSCs and HMSCs.

- **UML Sequence Diagrams**. UML Sequence Diagrams [OMG03] can be seen as OO variants of the ITU-T standard language Message Sequence Chart (MSC). A sequence diagram represents the interaction among the different objects of a system.

  In UML 1.x such sequence diagrams were quite simple. A sequence diagram represents the interaction among the different objects of a system, where horizontal arrows represent the messages between the life-lines represented by vertical lines. Loop and alternative are constructs introduced in UML 2.0 called *interaction fragments.* This feature allows to concisely describe within one diagram a set of traces, which would otherwise require a number of diagrams.

Other improvements include nesting capabilities and extended control constructs, make UML 2.0 sequence diagrams and MSCs have more or less the same expressiveness.

A number of papers address the problem of formalizing UML sequence diagrams. In [CSB01] a formal translation of Sequence Diagrams into Petri Nets is proposed, based on the UML Collaborations package meta-model. A Similar approach was proposed by Bernardi et al. [BDM02] who proposed an automatic translation of Sequence Diagrams into Generalized Stochastic Petri Nets. Aredo [Are02] presented formal semantics of UML sequence diagrams using PVS (Prototype Verification System) [OSR95]. Tahir et al. [TSBC05] proposed an operational semantics for UML Sequence diagrams based on a relation of causality between the actions of emission and reception of messages. This semantics are claimed to avoid unnecessary scheduling constraints and so ease the elaboration of high level specifications.

- **UML Use Cases Diagram**. The UML use case diagram captures Jacobson's use cases [Jac04]. A use case diagram describes a collection of use cases (given by a graphical and textual description) and external actors that interact with the system.

  [Lar01] distinguish two types of use cases:

  - Essential Use Cases [CL99]: they are abstract, lightweight, relatively free of technology and implementation detail; design decisions are deferred and abstracted, especially those related to the user interface [Lar01]. Essential use cases are of primary importance early in a project's analysis. Their purpose is to document the business process that the system must support without bias to technology.
  - Real Use Case: concretely describes the process in terms of its real current design, committed to specific input and output technologies, and so on. When a user interface is involved, they often show screen shots and discuss interaction with the widgets [Lar01].

UML use-case models supports reuse through four generalization relationships: (1) Extend dependencies between use cases, (2) Include dependencies between use cases, (3) Inheritance between use cases, and (4) Inheritance between actors. These relationships have given rise to a great deal of confusion because they do not have precise semantics. Génova et al. [GL04] revealed these ambiguities and imprecisions and they proposed a solution.

Overgaard et al. [ÖP98] provide an operational semantics of Use Case constructs and their specific types of relationships, namely Uses, Extends, Association, Dependency and Constraint, using an object-oriented specification language named *ODAL*, which has been formalized using the p-calculus [MPW92].

Operation schemas [SS00] have been proposed to formalize use cases based on the observation that use cases provide an informal description of interactions between a system and its actors, whereas operation schemas precisely describe a particular system action which executes atomically. Since operation schemas are more precise and formal than natural language, they offer some rigorous basis which makes some reasoning possible.

Shen et al. [SL03] provided a new formal language (High-level Constraint Language)(HCL) [SGY$^+$04] which can be used to describe the pre- and post- condition for a use case. Furthermore, if the pre- and post- condition in a use case diagram are executable, then the resulting model can be translated to *AsmL* language which allows for requirement model execution. The resulting HCL model can be tested and simulated.

- **UML Activity Diagrams.** Activity Diagrams capture the dynamic behavior of a system within the UML framework. The purpose of the activity diagram is to model a workflow process and/or to model operations.

  UML 2.0 has introduced significant changes and additions to activity diagrams. One of the most novel concepts introduced, are so called structured nodes (StructuredActivityNodes in the meta-model). This concept includes features like loops, expansion regions, collection valued parameters, and data streaming. UML 2.0 activity diagrams are typically used for business process modeling, for modeling the logic captured by a single use case or usage scenario, or for modeling the detailed logic of a business rule. In many ways UML activity diagrams are the object-oriented equivalent of flow charts and data flow diagrams (DFDs) from structured development.

  Activity diagrams share many characteristics with UCMs: focus on sequences of actions, guarded alternatives, and concurrency; complex activities can be refined; and simple mapping of behavior to components can be achieved through vertical swimlanes. However, activity diagrams do not capture dynamicity well, they do not support time constructs, and the binding of actions to components is semantically weak in the current UML standard.

  Many approaches to formalize Activity Diagrams have been proposed. Early work deals with formalizing UML1.x activity diagrams: Böerger et al. [BCR00a] propose a semantics of UML activity diagrams in terms of Abstract State Machines. The authors in [YsZ03] present a π-calculus [MPW92, Mil89] semantics for UML 1.4 activity diagrams. Eshuis and Wieringa [EW04] define two semantics approaches: requirements-level [EW01] and an implementation-level semantics [Esh02] for activity diagrams, intended for workflow modeling. These two semantics are based upon the STATEMATE semantics of Statecharts. Activity diagrams semantics have been also given in terms of Petri nets [GGW98].

  An almost complete formalization of UML2 activity diagrams using Petri nets is described by Störrle. He applies procedural Petri nets to formalize control flow [Sto04a], data flow [Stö05], exceptions and structured nodes [Stö04b]. Recently, Stöorrle and Hausmann [SH05] have identified several problems when formalizing UML 2.0 activity diagrams with Petri nets. They have shown that in principal, a Petri net formalization suffers from several problems such as inadequate support for streaming and traverse-to-completion.

  The authors in [VK05] proposed a formal semantics of a subset of UML 2.0 Activity Diagram relevant for business process modeling. They defined an Activity Diagram Virtual Machine (ADVM) based on the token flow (Petri net like) semantics. The Activity Diagram Virtual machine is defined by means of a meta-model, with operations defined by a mix of pseudocode and OCL pre- and postconditions.

- **Chisel Diagrams.** The CHISEL notation [AGG$^+$98] was developed by Bellcore (now Telcordia Technologies) as an informal graphical notation for describing telecomms services and features. Chisel notation has been used in the first feature interaction detection contest [GBGO00] and it demonstrated that it is capable of describing a wide variety of features.

  CHISEL diagrams are directed cyclic graphs that describe the sequences of events taking place on component interfaces. A diagram has numbered event nodes that contain input or output signals (but not both). Multiple signals in a node may be processed independently in parallel. Event nodes are linked by arcs which can be labeled with a boolean condition as a guard on the occurrence of a transition. Multiple abstract scenarios and actors can be involved, but internal actions are not covered. Decomposition is partially supported through references, and there is no language construct for the explicit support of time. CHISEL is supported by the Sculptor tool developed at Bellcore.

  Turner [TUR00] proposed an enhanced version of CHISEL with tightly defined rules for the syntax and static semantics of diagrams. The resulting notation is called CRESS (CHISEL Representation Employing Systematic Specification). CRESS has been formalized using SDL (Specification and Description Language) [IT02a] and LOTOS (Language Of Temporal Ordering Specification) [ISO89].

- **UML State Diagrams.** A UML state diagram is used to describe a system behavior in terms of its events and state changes. Its notations and semantics are substantially those of Harels Statechart [Har87, HP98] except it is an object-based variant of Harels.

  A UML State Diagram specifies the states a system may reside in and the transitions from one state to another. In addition, it also specifies what causes activities to start and stop, and how the system responds to various triggering events (An event may be generated by the system or by the environment). One statechart describes the behavior of a single class of objects. Statecharts may be synthesized using sequence, alternative, iteration and concurrency [ZHJ04].

  Much work has been carried out to give a formal semantics to the UML State Diagrams. These include those of J.Lilius et al. [Pal99] and D.Latella et.al [LMM99] which translate UML State Diagrams to Promela/SPIN that allow linear temporal logic model-checking. The works in [SZ02, LS02] formalize UML State Diagrams in B, [BCR00b] in ASM, [NB03, YLWD05] in CSP and [vdB01] uses labeled transition systems [NB03]. Jansamak and Surarerks [JS04] proposed transformation rules for formalizing UML statechart diagrams in terms of Concurrent Regular Expressions (CREs) [GR92]. Concurrent Regular Expressions are extensions of regular expression with four operators- interleaving, interleaving closure, synchronous composition and renaming. [CD05] presents a comparative literature survey on approaches to formally capture the semantics of UML state machines; it categorizes and compares 26 different approaches. Michael von der Beeck [vdB94] provides a comparison of 24 statecharts variants base on 19 criteria.

- **Live Sequence Charts (LSCs).** Live Sequence Charts (LSCs) [DH01] are introduced by Damm and Harel to overcome the shortcomings of MSCs. MSCs do not provide means to distinguish mandatory and possible behavior. LSCs introduce the distinction between scenarios that must happen, scenarios that may happen and scenarios that should never happen (i.e. negative scenarios). LSCs distinguish conditions that must be fulfilled (called *hot* conditions) from conditions that may be fulfilled (called *cold* conditions). Furthermore they provide means to specify an activation scenario, called *prechart*, which if successfully executed, forces the system to satisfy the scenario given in the actual chart body.

  [HM01, MHK02] extend the initial LSC version with assignments (which allow values of object properties or functions to be stored at one point in a chart), loops (which provide means for iteration within a single chart), variables (which allow for symbolic representation of the information exchanges between objects and thus to represent several instantiations of the same scenario with different actual values for each one) and symbolic object instances (which allow for the instances to be symbolic and parameterized). These extensions are part of a *play-in/out* methodology [HM01], which is supported by a tool, called the play-engine [HM01].

  LSCs have been applied to the automotive, telecommunication, and hardware domains [BGS05, CHK05]. The semantics of LSCs is briefly discussed in [DH01] using skeleton automata and program-like pseudo-codes. However, the first executable semantics for LSCs is described in [HM03]. Bontemps and Heymans [BH02] use Büchi automata to define High-level LSCs, a language expressed by a set of LSCs, so that standard algorithm for automata can be used to check consistency and refinement. Klose and Wittke [KW01] derive a similar timed Büchi automaton to capture the semantics of an LSC chart in isolation. [SD05] propose formal semantics of LSCs in terms of CSP.

- **Petri Nets (PNs).** Petri Nets were invented in 1962 by Carl Adam Petri in his Ph.D thesis [Pet62]. Petri Nets language [Pet77] is a graphical and mathematical modeling language used to capture functional requirements (sequential, alternative, asynchronous, distributed, parallel, non deterministic and concurrent scenarios) in a component-independent environment. It is also a formal specification technique with powerful methods for qualitative and quantitative analysis [Mur89].

  PNs consists of places, transitions, and directed arcs. Arcs run between places and transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. Places may contain any number of tokens. A distribution of tokens over the places of a net is called a marking. Transitions act on input tokens by a process known as *firing*. A transition is enabled if it can fire, i.e., there are tokens in every input place. When a transition fires, it consumes the tokens from its input places, performs some processing task, and places a specified number of tokens into each of its output places.

  Petri Nets have been used in a large variety of different areas. Their application ranges from informal to formal systems and from software to hardware systems and from sequential to concurrent systems. Petri Nets are used in communication protocols, distributed algorithms, computer architecture, computer organization, human-machine interaction and many others areas. Various kinds of Petri net classes with numerous features and analysis methods have been proposed in literature for different purposes and application areas. Amongst these extensions we find:

- Colored Petri Nets (CPNs) [Jen94, Jen92]. CPNs introduces the notion of token types, namely tokens are differentiated by colors, which may be arbitrary data values. Each place has an associated type, determining the kind of data that the place may contain.
- Hierarchical Colored Petri Nets (HCPNs) [HJS91]. HCPNs introduce a facility for building a PN out of subnets or modules. The idea behind the HCPNs theory is to allow the construction of a large model by using a number of small PNs, which are related to each other in a well-defined way.
- Object Petri Nets (OPNs) [Lak94]. OPNs are presented as an extension of colored Petri Nets made in a similar way to Hierarchical colored Petri Nets, but in an object-oriented perspective. OPNs support a complete integration of object-oriented concepts into Petri Nets, including inheritance and the associated polymorphism and dynamic binding. A class is defined as a Petri net, which can be, as usual, instantiated. In addition to places and transitions, a class contains data fields and functions. Data fields have types that may be simple (integer, real Boolean), class, or multi-set, which generalizes classical Petri net, places. New functions can be defined assuming predefined types and functions. Petri Net has at least, two-object-oriented extensions:(1) LOOPN (Language for OO Petri-Nets) and LOOPN++ [LK91] (2) COOPN (Concurrent OO Petri-Nets) and CO-OPN/2 [Bib97].
- Petri Nets with Time. The two main extensions of Petri Nets with time are *Time Petri Nets* (TPNs) [Mer74] and *Timed Petri Nets*(TdPNs) [Ram74]. TPNs associate with each transition a time interval. A transition can be fired if its enabling duration lies in its interval and time can elapse only if it does not disable some transition: firing of an enabled transition may depend on other enabled transitions even if they do not share any input or output place, which restricts a lot applicability of partial order methods in this model. Moreover, with this urgency requirement, all significant problems become undecidable for unbounded TPNs. Timed Petri Nets (TdPN), also called timed-arc Petri Nets, associate with each arc an interval (or bag of intervals). In TdPNs, each token has an age. This age is initially set to a value belonging to the interval of the arc which has produced it or set to zero if it belongs to the initial marking. Afterwards, ages of tokens evolve synchronously with time. A transition may be fired if tokens with age belonging to the intervals of its input arcs may be found in the current configuration.

High-Level Petri Nets [ISO04] have been standardized by ISO/IEC.

- **CREWS.** CREWS (Cooperative Requirements Engineering With Scenarios) is a RE research project funded by the European Community (21.903) of the ESPRIT framework programme.

Its goal is to develop, evaluate, and demonstrate the applicability of methods and tools for cooperative scenario-based requirements elicitation and validation. Natural language is used to elicit stakeholders requirements (i.e. textual scenarios) [RA98]. These scenarios are validated by the use of cooperative requirements animation, and by systematic comparison of the specification with usage test scenarios. This notation is supported by a tool called L'Ecritoire [RAC+98]. The CREWS-LEcritoire approach aims at eliciting requirements through a bi-directional coupling of goals and scenarios. The result will be Requirement Chunks (RC) which are pairs $\langle G$, $Sc\rangle$ where G is a goal and Sc is a scenario. A requirement chunk is a possible way of achieving a goal. Requirement Chunks (RCs) are related through:

- Composition: AND relationships among RCs link together those chunks that require each other to define a completely functioning system.
- Alternative: RCs related through OR relationships represent alternative ways of fulfilling the same goal.
- Refinement relationships: Relates requirement chunks at different levels of abstraction.

Dardenne et al.(KAOS Approach) [DvLF93] use a set of predefined levels of abstraction to link high level goals and operational requirements. However, goals are operationalized with logical predicates.

- **The Behavior Tree Notation.** The Behavior Trees (BT) notation [Dro03] is a graphical notation to capture the functional requirements of a system (provided in natural language) in a simple tree-like form. A behavioral tree is composed of nodes and edges. A node may be one of the following five types: a state realization; a selection (or condition); a guard; an internal event modeling communication and data flow between components within the system; an external event modeling communication and data flow with the environment of the system. A node refers to a particular component, C, and a behavior, B. In addition, each node can be labeled by one or more flags. A flag can specify:(a) a reversion in case the node is a leaf node, indicating that the control flow loops back to the matching node (i.e., a node with same component name, type and behavior);(b) a macro node, indicating that the flow continues from the matching node;(c) killing of a thread, which kills the thread that starts with the matching node, or (d) a synchronization point, where the control flow waits until all other threads with a matching synchronization point have reached the synchronization point [LKR07]. Dromey [Dro03] claims that BTs provide a direct and clearly traceable relationship between what is expressed in the natural language representation and its formal specification. Translation is carried out on a sentence-by-sentence basis.

  Individual requirement behavior trees (RBTs) for individual functional requirements are integrated, one-at-a-time, into an evolving design behavior tree (DBT). Integration of requirements trees [Dro03, WD04] is carried out on the graphical level. An RBT is merged with a DBT if its root node matches one of the nodes of the DBT. This process is called *genetic design process* [DP05]. Semantically, the merging step takes place when the matching node provides the point at which the preconditions of the merged RBTs are satisfied. Kirsten [Win04] developed a formal semantics for a subset of Behavior Trees using CSP.

The notation, as introduced by Dromey [Dro03], does not support the concept of time and consequently its application is limited to non-real-time systems. Lars et al. [LKR07] have recently extended it to include timing constraints.

- **Use Case Maps (UCMs).** The Use Case Maps notation [BC96] is a high level scenario based modeling technique that can be used to specify functional requirements and high-level designs for reactive and distributed systems. UCMs are expressed by a simple visual notation that allows for an abstract description of scenarios in terms of causal relationships between responsibilities (e.g. event, operation, action, task, function, etc.) along paths allocated to a set of components. These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g. post-conditions and resulting events). In UCM, scenarios are expressed above the level of messages exchanged between components, hence, they are not necessarily bound to a specific underlying structure (these types of UCMs are called Unbound UCMs). Components are also generic and can represent software entities (objects, processes, databases, servers, functional entities, network entities, etc.) as well as non-software entities (e.g. users, actors, processors). UCMs can also capture run-time behavior through dynamic stubs and dynamic responsibilities, and they have partial support for time constructs with timers and time-out paths. Concrete scenarios can be extracted using a simple path data model (Boolean variables) and scenario definitions, where initial values and triggered start points are provided. For a detailed description of many aspects of the UCM notation the reader is referred to Section 2.1.

- **Scenario Trees.** Hsia et al. [HSG+94] describe user oriented scenario trees that represents all scenarios for a particular user. Scenario trees are composed of nodes, which capture system states, and of directed arcs representing events that allow the transition from one state to the next. The scenarios are created by tracing events from the node at the top of the tree through a unique path to a terminal node on the bottom of the tree. All the scenarios are logically associated with the user views, and a formal BNF like grammar is created for each user view. This approach is effective when applied to a single thread of control and well-defined state transition sequences that have few alternative courses of action and no concurrency. However, this is not the case for industrial application especially telecommunication systems.

- **Somé's Scenarios.** Somé et al. [SDV95, SDV96] describe a scenario as a sequence of operations and time of occurrence, that may depend on conditions in the system and environment. From a user point of view, two kinds of operations may be distinguished in a scenario: actions on a system interface (stimuli), and reactions to them. The time of occurrence of operations can be constrained by interaction *initial delays* and *timeouts* and scenario *timeouts*. An interaction *initial delay* specifies a minimal, a maximal or an exact amount of time that must pass between the interaction first operation, and the last operation of the interaction preceding it. This notation is implemented in the Use Case Editor tool (UCEd)[Som04]. Furthermore, Somé et al. [SDV95, SDV96] have provided a formal interpretation of a scenario as a quadruple $\langle R_{num}, R_p, R_I, R_D \rangle$ where R_{num} is a scenario number, R_P is the scenario precondition, R_I is a sequence of interactions and R_D is a scenario timeout. The resulting scenarios are then translated into a timed automata specification.

3.2 Scenarios Integration

In the area of scenario integration, most research has only addressed the problem of sequential integration [DFKM98, SDV95, KM94b], and few researchers have been interested in a more general form of integration which consider composition of state machines synthesized from scenarios.

Desharnais et al. [DFKM98] defines a scenario as the union of two relations Re and Rs where Re represents the relation of the environment which captures all the possible actions of the environment and Rs the relation corresponding to the system reaction. The scenario integration is given by the composition of the scenarios relations. Koskimies and Makinen [KM94b] presents an algorithm for synthesizing a Statechart for an object of a system from a list of scenarios. They infer a Statechart that is able to execute all traces corresponding to the input scenarios. Somé et al. [SDV95] proposed sequential, alternative and parallel composition of textual timed scenario. Dano et al. [DBB97] proposed a formalization of use cases with Petri nets, the authors defined a list of temporal relations between use cases (begin at the same time, end at the same time, one after the other, etc.). Klein et al. [KCH05] proposed a merge operator for behavioral requirements expressed by Message Sequence Charts and showed how this product can be systematically used to integrate new behaviors in an existing one. Glinz [Gli95] presented a way for composing scenarios represented by Statecharts using some operators (conditional, iterative and concurrent), but without supporting scenarios overlapping.

Standard scenario-based modeling diagrams such as UML2 Interaction Overview Diagrams(IOD) [OMG05] and ITU high-level MSCs [IT04] address the scenario integration problem. These diagrams are essentially graphs whose nodes represent scenarios and edges show the control flow between them. Behaviors specified by nodes are considered as non-overlapping.

Use Case Maps addresses inter-scenario overlapping but does not provide an integration algorithm.

3.3 Classifications of Scenario Notations

Many classification approaches have been proposed to categorize and compare scenario notations.

- Amyot et al. [AE03] define nine criteria to categorize and compare fifteen scenario-based notations. The proposed criteria are:
  - **Component focus:** Scenarios can be described in terms of communication events between system components or independently from components, in a pure functional style.

  - **Hiding:** Scenarios could describe system behavior with respect to their environment only (black-box), or they could include internal (hidden) information as well (gray-box).
  - **Representation:** Scenarios can be described in various ways, for instance with semi-formal pictures, natural language, structured text, logic, grammars, trees, state machines, tables, visual paths, and sequence diagrams.
  - **Ordering:** Scenarios represent a collection of events that can be ordered sequentially or causally.
  - **Time:** Support for expressing time constraints with appropriate data types and evaluation mechanisms.
  - **Decomposition:** Decomposition can be hierarchical (which improves scalability) or be achieved through dependencies (e.g. references, contains, etc.).
  - **Abstraction:** An abstract scenario is generic, with formal parameters, whereas a concrete scenario focuses on one specific instance, with concrete data values.
  - **Identity:** Scenarios can focus on one actor (useful for component-oriented implementations) or target many actors at once (useful when describing end-to-end situations).
  - **Dynamicity:** A scenario notation is dynamic when it enables the description of behavior that modifies itself at run-time, otherwise it is said to be static.

- The authors in [LDD06] have presented a comparative survey of 21 approaches found in the literature based on two sets of comparison criteria. One set of criteria is for assessing approaches from a user's perspective. The other set of criteria compare the approaches from a more technical perspective, by focusing on the synthesis of scenario-based models into state-based models.
  - **Criteria Relevant From a User's Perspective:**
    * **Intended use.** Approaches are classified as intended for analysis only, or for both analysis and code generation.
    * **Source notation.** The choice of source notation (syntax and semantics) may influence the users' ability to describe scenarios with different levels of expressiveness as well as affect the synthesis algorithms.
    * **Support of composition mechanism.** By using composition mechanisms some scenario notations have the ability to express behavior of complex systems more comprehensively.
    * **Support of parallelism.** Parallelism is either implicitly supported by means of the underlying semantics or explicitly supported by means of parallel composition constructs. By supporting parallelism, scenario notations can describe reactive systems more realistically.
    * **Target notation.** The choice of a target notation is mainly influenced by the intended use as well as the previous experience of the designer.

    * **Model type.** This criterion is closely related to the intended use. For instance, an approach may concentrate on deriving a set of object state models, or try to generate one single global state model for the whole system.
    * **Synthesis path:** Scenario-based models are categorized into either basic scenarios (BS) without using any composition mechanisms, or global scenarios (GS) obtained through the composition of BSs. These scenario-based are synthesized into state-based models: object state machines (OSM) and global state machines (GSM) which are composed of OSMs. Four synthesis paths from scenario-based models to state-based models were proposed: BS$\rightarrow$ OSM, BS$\rightarrow$GSM, GS$\rightarrow$OSM, and GS$\rightarrow$GSM).
    * **Degree of automation:** The generation of state-based models from scenario-based models can either be semi-automatic or fully-automatic.
    * **Tool support:** We list whether a synthesis approach is supported by a tool.
  - **Criteria Relevant From a Technical Perspective:**
    * **Inter-scenario relationships.** The authors have identified five different ways to identify the Inter-scenario relationships. The designers can implicitly infer the relationships from the scenarios by using events or from the semantics of scenario notations. Additionally, the designers can explicitly define the relationships among scenarios by composition mechanisms, conditions, or a combination (hybrid) of both composition mechanisms and conditions.
    * **Consistency check.** Approaches may allow checking the consistency of scenario-based models before or during the synthesis processes.
    * **Completeness check.** Completeness checks on implied scenarios (extra behaviors) or missing scenarios (fewer behaviors) may be provided by the approaches.
    * **State space reduction.** Depending on how the inter-scenario relationships are identified, an approach may merge states and thus reduce the state space to various degrees.

- Rolland et al. [RAC+98] propose a scenario classification based on four views:
  - *Content View*: what part of the work activity is captured in a scenario ?
  - *Form View*: How is it represented in the development environment ?
  - *Purpose View*: For what usage in the design process is it captured (purpose view) ?
  - *Life-cycle view*: How is it developed and evolved ?

- Cockburn [Coc97] uses four dimensions to use case descriptions, namely purpose, content, plurality, and structure. Purpose can be either for stories (explanations) or for requirements. Content can be contradicting, consistent prose, or formal content. Plurality is either one or multiple, in a way similar to multiplicity. Structure can be unstructured, semi-formal, or formal.

- In a European industrial survey, Arnold et al. [AEG+98] proposed a classification taxonomy for scenarios usage in industrial projects. Their criteria are grouped under five main divisions: project properties, scenario contents and representation, goals, process, and experiences and expectations. They surveyed twelve industrial projects from various domains (telecommunications, sales, medical, software development, insurance, banking) where scenarios are used.

- Chance and Melhart [CM99] introduced a taxonomy of scenarios with the objective to improve understanding of scenarios and their usage. This taxonomy is organized into a hierarchy according to the purpose of the scenario. Scenarios can describe basic functionality (operational scenario), describe abnormal conditions (failure scenario), help evaluate system response (performance scenario), aid in requirements analysis and elicitation (refinement scenario), or be used to explain the system behavior to others (learning scenario). Each type of scenario is described in terms of key attributes: (1) Description: This describes how this category of scenarios differs from all other categories; (2) Creators/Users: This lists the architects and developers most likely to create or use the scenario category; (3) Information Needed: This lists the information that is useful to create scenarios of this category; (4) Uses: This lists the most common uses of scenarios of this category.

3.4 Formal Semantics

Formal semantics is concerned with the rigorous mathematical study of the meaning of programming languages and models of computation. Formal approaches use mathematical and logical techniques to more precisely define language semantics. There are three basic approaches:

- **Axiomatic semantics** [Hoa83] is an approach based on mathematical logic to proving the correctness of computer programs. Axiomatic semantics provide an abstract semantics definitions of the language entities and their relations to each other in terms of axioms that are concise and understandable. Specific properties of the effects of executing the constructs are expressed as assertions - predicates with variables, where the variables define the state of the program. However, axiomatic semantics remain very complex for real languages (i.e., large descriptions for many basic constructs) and they has little or no guidance to tool developers.

- **Denotational semantics** [Mos90] consist on translating an expression from the language into an expression in some mathematical domain (i.e., usually functions). This mapping allows for formal manipulation and deriving properties. Denotational semantics build on known domains using syntactic structures, however it is still too complex for users.

- **Operational semantics** [Plo81] consists of a procedure to transform an expression into a behavior (an execution step) and a new expression (the result after executing this step). Informally, the goal of an operational semantics is, given an expression denoting a process in a certain state, to describe all possible activities that can be performed by the process in that state and to describe the state of the process after such an activity. An operational semantics may be easier to understand than an equivalent denotational one, because the computational model can be more intuitive than the abstract mathematical components of a denotational semantics. In this thesis, we focus on formal operational semantics.

3.5 Formal Description Techniques

Formal methods are mathematical specification languages with formal syntax and semantics, which offer rigorous support of system development leading to the early detection of errors. They focus on reliability and correctness of systems by using formal validation and verification techniques. The benefits of formal methods have been discussed many times in the literature supported by the existence of a significant body of research in the area. Nevertheless, its industry and end-user acceptance remain limited in practice since the tradeoff between productivity and reliability is still too large.

Abstract State Machines, LOTOS, SDL, MSC and Petri Nets represent five formal specification techniques that are particularly relevant to the scenario-based description of high-level requirement specifications. While MSC and Petri Nets (PN) have been introduced in Section 3.1 and Abstract State Machines(ASM) in Section 2.2, this section provides a brief overview of LOTOS and SDL.

3.5.1 LOTOS

LOTOS (Language of Temporal Ordering Specification) [ISO89], an algebraic specification language, was developed by the FDT experts of the working group ISO/TC97/SC21/WG1 during the 80s. It is a specification language developed for the formal description of the various elements of the OSI (Open System Interconnection) architecture such as services and protocols. Nowadays, the LOTOS application area has been applied extensively in both universities and industry to cover various domains such as distributed and concurrent systems in general.

The basic idea of LOTOS is that systems can be specified by defining the temporal relations among the actions that constitute the system externally observable behavior. LOTOS language consists of:

- A control component in which LOTOS behavior expressions are described. It is based on Milners Calculus of Communicating Systems (CCS) [Mil89] and Hoare's Communicating Sequential Processes (CSP) [Hoa85], which include the concepts of action prefix, choice, parallel composition, multi-way synchronization, hiding, process instantiation, and a few others.

- A data type component, which is based on the formal theory of algebraic abstract data types *ACT ONE* [EM85]. It deals with the description of data structures and value expressions.

LOTOS is suitable for the integration of behavior and structure in a unique executable model. LOTOS allows the use of many tool-supported validation and verification techniques such as CADP (CÆSAR-ALDEBARAN Distribution Platform) [RL07a], ELUDO (Environnement LOTOS de l'Universite D'Ottawa) [Gar96] and LOLA (LOtos LAboratory) [PN91]. A number of excellent LOTOS tutorials exist in the literature [LFHH91, BB87].

Daniel Amyot [Amy01a], in his PhD thesis, claims that Use Case Maps and LOTOS represent a good match and presents several factors that motivate the choice of LOTOS as a formal framework for Use Case Maps. In Section 3.6, we give a brief introduction to SPEC-VALUE [Amy01a], an iterative and incremental scenario-driven approach for the description and validation of complex system functionalities at the early development stages using Use Case Maps and LOTOS.

3.5.2 SDL

The Specification and Description Language (SDL) [IT02a] is an object-oriented, formal language defined by The International Telecommunications Union-Telecommunications Standardization Sector (ITU-T) (formerly Comitè Consultatif International Telegraphique et Telephonique [CCITT]) as recommendation Z.100. The language is intended for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. SDL covers different levels of abstraction, from a broad overview down to detailed design. The basic theoretical model of an SDL system consists of a set of extended finite state machines (EFSMs) that run concurrently. These machines are independent of each other and communicate by means of asynchronous discrete signals and synchronous remote procedure calls. Both mechanisms can carry parameters to interchange and synchronize information between SDL processes and their environment. He et al. [HAW03] present an approach to synthesize SDL models from MSCs generated from UCM specifications.

3.6 UCM Validation: SPEC-Value Approach

SPEC-VALUE [Amy01a] is an iterative and incremental scenario-driven approach for the description and validation of complex system functionalities at the early development stages. Functional requirements are captured using UCM notation. The responsibilities defined in the UCMs are then allocated to the components in the selected underlying structure. Then the UCM scenarios are translated into detailed LOTOS specifications that are validated with the help of tools. The validation testing approach introduced in SPEC-VALUE proposes the generation of test cases at the design stage from the information provided by the users' requirements. Figure 3.1 illustrates the SPEC-VALUE approach.

The authors claim that the gap between UCM and LOTOS is small, and a translation from UCMs to LOTOS is straightforward. The synthesis of the LOTOS specification from UCMs is performed manually. Later, Guan [Gua02] provided a synthesizer for the generation of LOTOS models from UCMs. Her work automates many of the construction rules proposed in SPEC-VALUE. The SPEC-VALUE approach follows a well-established method of mapping a semi-formal language to a formal language. However, mapping requires a manual verification step to decide about the completeness and details of the resulting LOTOS specification. While there is no "incorrect" semantics, inconsistent mapping rules can introduce unexpected behavioral consequences.

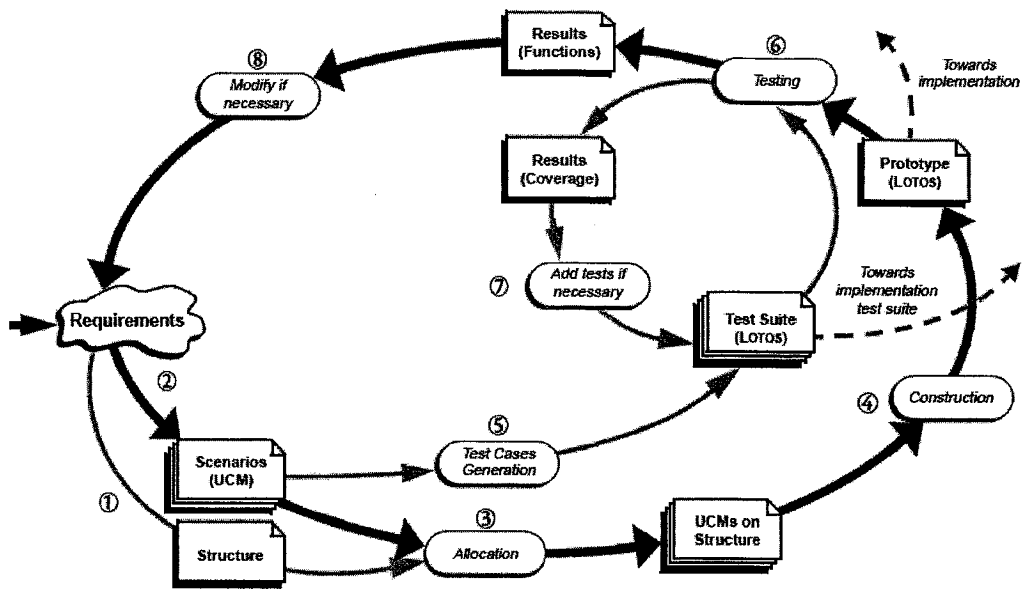


Figure 3.1: Specification-Validation Approach with LOTOS and UCMs (SPEC-VALUE) [Amy01a]

3.6.1 Testing Approach in SPEC-VALUE

The testing approach in SPEC-VALUE is illustrated in Figure 3.1. Since LOTOS is the targeted FDT for SPEC-VALUE, LOTOS test cases are generated manually from UCMs (step 5). Abstract sequences of actions are extracted from unbound UCMs and transformed into LOTOS test processes. The testing is performed by composing the test cases with the LOTOS prototype (step 6). This operation is performed automatically by using a LOTOS testing tool called LOLA, which then outputs the resulting verdict for each test (i.e. *pass*, *may pass* or *fail*). If a verdict is not satisfactory, then appropriate modifications might be brought to the requirements (step 8), which may result in cascading modifications to the scenarios, the tests and the prototype.

The testing approach in SPEC-VALUE is claimed to be *validation testing* rather then *conformance testing*. In SPEC-VALUE, the test suite is derived from informal requirements and semi-formal scenarios (the UCMs), and the goal is to create and check the LOTOS specification model (prototype). This test suite is used to validate the model against the requirements, hence the term validation. Conformance testing can be used at a later stage of the design cycle, when an implementation is required to be declared conformant to the formal model.

3.6.2 UCM-Oriented Testing Patterns

Daniel Amyot [Amy01a] proposed eight UCM-oriented testing patterns to cover alternatives, concurrent paths, loops, multiple start points, single stubs, and causally linked stubs. These patterns aim to cover functional scenarios at various levels of completeness: all results, all causes and all results, all path segments, all end-to-end paths, all plug-ins, and so on. The patterns are inspired partly from various existing test selection strategies for implementation languages constructs such as branching conditions and loops. Hence, this UCM-based test selection shares many concepts with white-box testing. The author claims that the proposed set of patterns can be applied to a multitude of contexts, and can be combined together (e.g. in a pattern language) for dealing with complex UCMs. The proposed set of patterns has been revisited and refined later in [ALW05].

3.7 Chapter Summary

This chapter reviews existing work and concepts in four areas of interest to *Early Stages V&V approach*. In Section 3.1, we have covered many high level scenario based notations. Section 3.2 discussed scenario integration approaches and Section 3.3 presented existing classification approaches. Section 3.4 focused on types of formal semantics while Section 3.5 discussed formal specification techniques especially LOTOS, which is used in SPEC-VALUE approach [Amy01a] and SDL which is largely used in the telecommunication domain. Other formal description techniques such as MSC and Petri Nets (PN) have been introduced in Section 3.1 while Abstract State Machines(ASM) have been introduced in Section 2.2. Finally, in Section 3.6, we have presented briefly the SPEC-VALUE approach [Amy01a].

Chapter 4

An ASM Operational Semantics for Use Case Maps

In this chapter[1], we present a formal operational semantics for Use Case Maps language based on Abstract State Machines. We present two possible ASM-based solutions: (1) Multi-Agent ASM solution and (2) Single-Agent ASM solution with non deterministic interleaving. Our ASM model provides a concise semantics of UCM functional constructs and describes precisely the control semantics.

4.1 ASM-based UCM Formal Syntax

4.1.1 Use Case Maps Formal Syntax

Before defining the ASM formal semantics of UCM specifications, we define a UCM specification as follows:

Definition 1 (Use Case Maps.) *We assume that a UCM specification is denoted by a 7-tuple (D, H, λ, C, GVar, B_c, B_s) where:*

- *D is the UCM domain, composed of sets of typed elements. D= $SP \cup EP \cup R \cup AF \cup AJ \cup OF \cup OJ \cup Ts \cup Tm \cup ST$. Where SP, EP, R, AF, AJ, OF, OJ, Ts, Tm and ST are respectively the sets of Start Points, End Points, Responsibilities, AND-Fork, AND-Join, OR-Fork, OR-Join, Time Stamps, Timers and Stubs.*
- *H is the set of edges connecting UCM constructs to each other.*
- *λ is a transition relation defined as: $\lambda = D \times H \times D$.*
- *C is the set of components.*

[1]This chapter content is published in IEEE International Conference on Requirements Engineering - RE 2005 [HRD05b] and in Formal Techniques for Networked and Distributed Systems - FORTE 2005 [HRD05a]

- *GVar is the set of global variables.*
- *B_c is a component binding relation defined as $B_c = D \times C$. B_c specifies which element of D is associated with which component of C. B_c is empty for unbound UCM.*
- *B_s is a stub binding relation and is defined as $B_s = ST \times IN/OUT \times SP/EP$. B_s specifies how the start and end points of the plug-in map would be connected to the path segments going into or out of the stub.*

The definition of the ASM formal semantics of UCM consists of associating each UCM construct with an ASM, which models its behavior. In this section, we associate first an ASM signature with each UCM construct and then assign execution rules to them.

4.1.2 ASM Signature of UCM Constructs

The UCM maps are modeled using the abstract sets: *StartPoint, EndPoint, Responsibility, AND-Fork, AND-Join, OR-Fork, OR-Join, Stub* and *Timer.* The abstract set H represents the set of edges connecting UCM constructs.

- **Start Points** are of the form *StartPoint(PreCondition-set, TriggerringEvent-set, StartLabel, in, out)*, where the parameter *PreConditions-set* is a list of conditions that must be satisfied in order for the scenario to be enabled (if no precondition is specified, then by default it is set to true). The parameter *TriggeringEvents-set* is a list that gives the set of events that can initiate the scenario along a path. One event is sufficient for triggering the scenario. The parameter *StartLabel* denotes the label of the start point. A start point should not have an incoming edge except when connected to an end point (called a waiting place). In such situation, we use the parameter $in \in H$ to represent the connection with an end point. The parameter $out \in H$ is the (unique) outgoing edge.
- **End Points** are of the form *EndPoint(PostCondition-set, ResultingEvent-set, EndLabel, in, out)*, where the parameter *PostConditions-set* is a list of conditions that must be satisfied once the scenario is completed. The parameter *ResultingEvent-set* is a list that gives the set of events that result from the completion of the scenario path. The parameter *EndLabel* denotes the label of the end point; the parameter $in \in H$ is the (unique) incoming edge. End points have no target edge except when connected to a start point (i.e. a waiting place). In such a case, $out \in H$ represents such connection.
- **Responsibilities** are of the form *Responsibility(in, Resp, out)*, where $in \in H$ is the incoming edge, *Resp* is the responsibility to be executed (to be defined by a set of simultaneous ASM function updates), and $out \in H$ is the outgoing edge. A responsibility is connected to only one source edge and to one target edge.
- **OR-Forks** are of the form *OR-Fork(in, $[Cond_i]_{i \leq n}$, $[out_i]_{i \leq n}$)*, where *in* denotes the incoming edge, $[Cond_i]_{i \leq n}$ is a finite sequence of Boolean expressions, and $[out_i]_{i \leq n}$ is a sequence of outgoing edges.

- **OR-Joins** are of the form *OR-Join*($\{in_i\}_{i\leq n}$, *out*), where $\{in_i\}_{i\leq n}$ denotes the incoming edges and, *out* is the outgoing edge.

- **AND-Forks** are of the form *AND-Fork*(*in*, $\{out_i\}_{i\leq n}$), where *in* denotes the incoming edge, and $\{out_i\}_{i\leq n}$ is a sequence of outgoing edges.

- **AND-Joins** are of the form *AND-Join*($\{in_i\}_{i\leq n}$, *out*), where $\{in_i\}_{i\leq n}$ denotes the incoming edges, and *out* is the outgoing edge.

- **Timers** are of the form *Timer(in, TriggerringEvent-set, out, out_timeout)*, where *in* denotes the incoming edge. The parameter *TriggeringEvents-set* is the list that defines the set of events that can trigger the continuation path (i.e. represented by *out*) and the parameter *out_timeout* $\in H$ denotes the timeout path.

- **Stubs** have the form *Stub*($\{entry_i\}_{i\leq n}$, $\{exit_j\}_{j\leq m}$, *isDynamic*, $[Cond_k]_{k\leq l}$, $[plugin_k]_{k\leq l}$) where $\{entry_i\}_{i\leq n}$ and $\{exit_j\}_{j\leq m}$ denote respectively the set of the stub entry and exit points. *isDynamic* indicates whether the stub is dynamic or static. Dynamic stubs may contain multiple plug-ins, $[plugin_k]_{k\leq l}$ whose selection can be determined at run-time according to a selection-policy specified by the sequence of Boolean expressions $[Cond_k]_{k\leq l}$. The sequence Cond is empty for static stubs (i.e. *isDynamic=false*).

4.2 ASM-based UCM Formal Semantics

In this section, we present two possible ASM-based solutions: (1) Multi-Agent ASM solution and (2) Single-Agent ASM solution with non deterministic interleaving.

4.2.1 Multi Agent ASM-based Solution

The current AsmL version does not support interleaving between concurrent agents. Each agent should run to completion before a new one can start executing. Before presenting the Multi-Agent ASM based rules that describe the operational semantics of UCM constructs, we present the abstract sets and functions necessary for encoding UCM specifications.

Access functions

For each UCM construct we use a (static) function *Param* which, when applied to a UCM construct (i.e., C($Param_1$,...,$Param_n$)) yields the parameter. For example *out(StartPoint)* yields the outgoing edge of the construct *StartPoint*, *in(Responsibility)* yields the incoming edge of the construct *Responsibility*, *Cond(OR-Fork)* yields the sequence of conditions of construct *OR-Fork*. We often suppress parameters notationally.

Encoding UCM Hierarchy

We formalize UCM maps by an abstract set *MAPS*. It contains the root map (i.e. the main UCM map) and all its submaps (i.e. plug-ins). The nesting structure of a UCM specification is encoded in the following functions:

- UpMap: MAPS $\rightarrow$ MAPS $\cup$ $\{undef\}$, assigns to a plug-in its immediately enclosing map, if any. We assume that this function yields *undef* for the root map which is not enclosed in any map. Thus, *UpMap(rootMap)=undef.*

- StubBinding:$\{\{entry_i\}\cup\{EndPoints\}\}$ $\times$ MAPS $\rightarrow\{\{StartPoints\}\cup\{exit_j\}\}$ specifies how a plug-in $\in$MAPS is bound to a stub. The path segments that are connected to the stub need to be bound to the paths of the plug-ins in order to express continuity. This is done through explicit binding. An entry edge joins a stub entry with a start point from the plug-in. An exit edge joins a stub exit with an end point from the same plug-in.

Agent Decomposition

Let AGENT be the abstract set of agents *a* which move through their associated UCM map, by executing the UCM construct at the current active edge, i.e. the edge where the agent's control lies. Each sequential UCM segment can be represented by an independent *agent.* Figure 4.1 shows a UCM with five agents (i.e., $Agent_1$, $Agent_2$... $Agent_5$). Every agent can mainly be characterized by three dynamic functions:

- active: AGENT$\rightarrow$ H represents the identifier of the active edge leading to the next UCM construct to be executed.

- mode: AGENT$\rightarrow$ $\{running, inactive\}$. An agent may be running in normal mode or inactive once the agent has finished its computation.

- level: AGENT$\rightarrow$MAPS provides the submap that the agent is currently traversing.

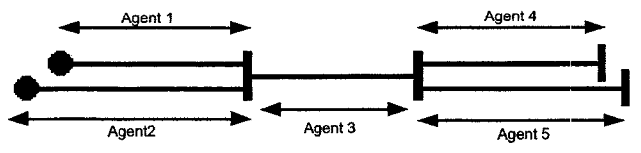


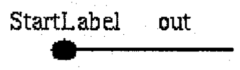
Figure 4.1: A UCM decomposition into five agents

For the root map, it is required that there is an agent for each starting point, in running mode with active edges positioned on the corresponding start points of the root map (i.e. *active=in(StartPoint))*. The creation of the initial ASM agents, their initialization and the initialization of the global variables used in the scenario definitions represent the initialization phase.

Typically, a running agent has to look at the target of its currently active edge to determine the next action. **me** refers to the current agent and *CurrConstruct* denotes the current UCM construct to be executed, i.e. the UCM construct where (**me**.active=in(construct)) $\wedge$ (**me**.mode=running).

ASM Rules of UCM Constructs

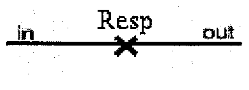
- **Start points.** If the control is on the edge *in(StartPoint)*, the *PreCondition-set* is satisfied and there occurs at least one event from the *triggeringEvent-set*, then the start point is triggered and the control passes to the outgoing edge of the StartPoint (Otherwise nothing happens and the control stays at the StartPoint). Figure 4.2 describes the start point rule.



```
if CurrConstruct is StartPoint(PreCondition-set, TriggerringEvent-set, StartLabel, in, out) then
  if (EvaluatePreConditions & EvaluateTrigger) then me.active:= out
where:
  - EvaluateTrigger: TriggerringEvent-set × {events} → Boolean; evaluates whether the set of events occurring at StartPoint are included in the TriggeringEvent-set.
  - EvaluatePreConditions: PreCondition-set → Boolean evaluates whether all preconditions are satisfied.
```

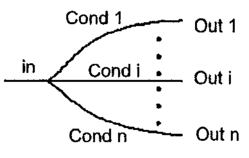
Figure 4.2: Multi Agent Solution: Rule of Start Point

- **Responsibilities.** If the control is on the edge *in(Responsibility)* then *Resp* is performed and the control passes to the outgoing edge. Figure 4.3 illustrates the responsibility rule.



```
if CurrConstruct is Responsibility (in,Resp,out) then
    Resp
    me.active:= out
```

Figure 4.3: Multi Agent Solution: Rule of Responsibility



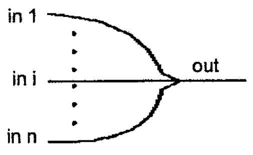
```
if CurrConstruct is OR-Fork(in, $[Cond_i]_{i\leq n}$,$[out_i]_{i\leq n}$)
then if NonDeterministicChoice($[Cond_i]_{i\leq n}$) then
            me.active:= (choose $out_i$ in $[out_k]_{k\leq l}$)
      else  if $Cond_1$ then me.active:=$out_1$
            ...
            if $Cond_n$ then me.active:= $out_n$
where NonDeterministicChoice:{Cond}→Boolean is a dynamic function that checks whether more than one condition evaluates to true and $[out_k]_{k\leq l}$ is the sequence of edges associated to satisfied conditions.
```

Figure 4.4: Multi Agent Solution: Rule of OR-Fork

- **OR-Fork.** If the control is on the incoming edge of an OR-Fork, the conditions are evaluated and the control passes to the edge associated to the true condition. If more than one condition evaluates to true (i.e. nondeterministic choice), the control passes randomly to one of the outgoing edges associated to the true conditions. Figure 4.4 illustrates the OR-Fork rule.

- **OR-Join.** When one or many flows reach an OR-Join, the control passes to the outgoing edge. Figure 4.5 illustrates the OR-Join rule.

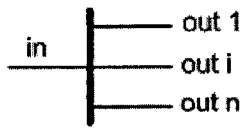
  Note: An UCM loop can be modeled as an OR-Fork followed by an OR-Join. Their respective rules should be executed once encountered.



```
if CurrConstruct is OR-Join($\{in_i\}_{i\leq n}$, out) then
    me.active:= out
```

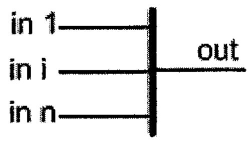
Figure 4.5: Multi Agent Solution: Rule of OR-Join

- **AND-Fork.** When the control is on an incoming edge of an AND-Fork synchronization bar, then the flow is split into two or more flows of control. The currently running agent creates the necessary new subagents and sets their mode to running, then sets its mode to inactive. Each new ASM subagent inherits the program for executing UCMs, and its control is started on the associated outgoing edge of the AND-Fork.



```
if CurrConstruct is AND-Fork(in, $\{out_i\}_{i\leq n}$) then
    me.mode:=inactive
    extend AGENT with $a_1, \ldots, a_n$
    do for all$a_i$, $1 \leq i \leq n$
     $a_i$.mode := running
     $a_i$.active := $out_i$
```

Figure 4.6: Multi Agent Solution: Rule of AND-Fork

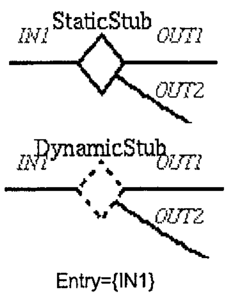


```
if CurrConstruct is AND-Join($\{in_i\}_{i\leq n}$, out)
then if not ($\forall a_1, \ldots, a_n$ $in_i = active(a_i)$) then
     me.mode:= inactive
   else   me.mode:= inactive
          extend AGENT with $a_{n+1}$
          $a_{n+1}$.active:= $out$
          $a_{n+1}$.mode:= running
```

Figure 4.7: Multi Agent Solution: Rule of AND-Join

- **AND-Join.** When many subagents running in parallel reach an AND-Join, their parallel flow must be joined. When all incoming edges become active, a new agent is created and the control passes to the outgoing edge. The last agent arriving at the AND-Join will fire the rule. Inactive agents are deleted after each rule's execution.

- **Stub**. Once the control reaches a stub, the control passes to the selected plug-in and the execution continues following the UCM semantics. No extra agents are needed to execute a *Stub* unless the selected plug-in contains a concurrent flow.



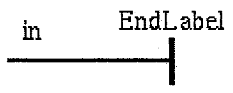
Entry={IN1}
Exit={OUT1,OUT2}

```
if CurrConstruct is Stub($\{entry_i\}_{i\leq n}$,$\{exit_j\}_{j\leq m}$, isDynamic, $[Cond_k]_{k\leq l}$,$[plugin_k]_{k\leq l}$) then
  if not(isDynamic) then add(plugin, me.level) to MapHierarchy
                  me.level := plugin
                  me.active := in(StubBinding($entry_i$, plugin)
  else  add(plugin, me.level) to MapHierarchy
        me.level := SelectionPolicy($(Cond_k)_{k\leq l}$))
        me.active := in(StubBinding($entry_i$ , SelectionPolicy( $Cond_k)_{k\leq l}$))
Where SelectionPolicy:{Cond} $\rightarrow$ $MAPS$ is the selection policy function.
```

Figure 4.8: Multi Agent Solution: Rule of Stub

- **End point**. When the control reaches an end point, four cases have to be considered, depending on whether the end point is connected to a start point (i.e. a waiting place) and whether it is inside a plug-in or part of the root map:

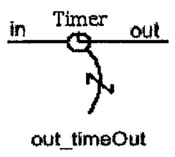
  1. If the end point is connected to a start point(i.e. a waiting place), then the control passes to the out edge.
  2. If the end point is inside a plug-in and it is bound to a stub, then the control passes to the stub's exit point bound to the plug-in end point.
  3. If the end point is inside a plug-in but it is not bound to a stub, then the running agent is stopped.
  4. If the end point is part of the root map (and not connected to a start point), then running agent is stopped.



```
if CurrConstruct is EndPoint(PostCondition-set, ResultingEvent-set, EndLabel, in, out)
then
  if (out$\neq$undef) then me.active := out
  else
    if UpMap(me.level)=undef) then me.mode:= inactive
    else
      if (StubBinding(EndPoint,me.level))$\neq$undef then
            me.active:= out(StubBinding(EndPoint, me.level))
      else me.mode:= inactive
```

Figure 4.9: Multi Agent Solution: Rule of End Point

The exit from nested maps should be performed in the correct order of the stub structure. However, one control may exit the stub while another one is still inside the stub.



```
if CurrConstruct is Timer(in, TriggerringEvent-set, out, out_timeout) then
    if (Triggered) then me.active:= out
    else me.active := out_timeout
where Triggered: TriggerringEvent-set→Boolean determines whether a trigger occurs within a predefined time frame.
```

Figure 4.10: Multi Agent Solution: Rule of Timer

- **Timer.** The timer rule is very similar to a basic OR-Fork rule with only two disjoint branches (out and out_timeOut).

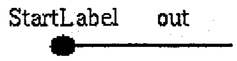
4.2.2 Single Agent Non Deterministic Interleaving Solution

The Multi-Agent solution assumes that each agent maintains a single active edge at every execution point(i.e., *a.active* is a singleton). In the single agent solution, we consider a unique thread of execution and a unique set of active edges (i.e., *a.active* is a set of edges). Contrary to the multi-agent solution, the single-agent solution ensures a full non-deterministic interleaving between concurrent threads.

ASM rules are modified to reflect the existence of one global set of active edges.

ASM Rules of UCM Constructs

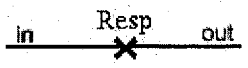
- **Start points.** If the control is on the edge *in(StartPoint)*, the *PreCondition-set* is satisfied and there occurs at least one event from the *triggeringEvent-set*, then the start point is triggered and the control passes to the outgoing edge of the StartPoint. The set of edges *active* is updated with the addition of edge *out* and the removal of edges *in.* Figure 4.11 describes the start point rule.



```
if CurrConstruct is StartPoint(PreCondition-set, TriggerringEvent-set, StartLabel, in, out) then
  if (EvaluatePreConditions & EvaluateTrigger) then add out to active
     remove in from active where:
  - EvaluateTrigger: TriggerringEvent-set × {events} → Boolean; evaluates whether the set of events occurring at StartPoint are included in the TriggeringEvent-set.
  - EvaluatePreConditions: PreCondition-set → Boolean evaluates whether all preconditions are satisfied.
```

Figure 4.11: Single Agent Solution: Rule of Start Point

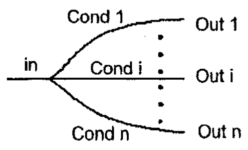
- **Responsibilities.** Responsibilities represent atomic actions, not to be decomposable, and their execution is not interruptible. If the control is on the edge *in(Responsibility)* then *Resp* is performed and the control passes to the outgoing edge. The set of edges *active* is updated with the addition of edge *out* and the removal of edges *in*. Figure 4.12 describes the responsibility rule.



```
if CurrConstruct is Responsibility (in,Resp,out) then
    Resp
    add out to active
    remove in from active
```

Figure 4.12: Single Agent Solution: Rule of Responsibility

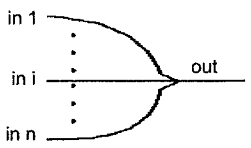
- **OR-Fork**. If the control is on the incoming edge of an OR-Fork, the conditions are evaluated and the control passes to the edge associated to the true condition. If more than one condition evaluates to true (i.e. nondeterministic choice), the control passes randomly to one of the outgoing edges associated to the true conditions. The set of edges *active* is updated with the addition of edge out_i that corresponds to the true condition and the removal of edges *in*. Figure 4.13 illustrates the OR-Fork rule.



```
if CurrConstruct is OR-Fork(in, $[Cond_i]_{i\leq n}$,$[out_i]_{i\leq n}$)
then add (choose k in $[out_k]_{k\leq l}$) to active
    remove in from active
```

Figure 4.13: Single Agent Solution: Rule of OR-Fork

- **OR-Join**. When one or many flows reach an OR-Join, the control passes to the outgoing edge. The set of edges *active* is updated with the addition of edge *out* and the removal of edges in_i from which the control reached the OR-Join. Figure 4.14 illustrates the OR-Join rule.

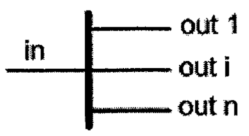


```
if CurrConstruct is OR-Join($\{in_i\}_{i\leq n}$, out) then
    add out to active
    forall k in $\{in_i\}$
    remove k from active
```

Figure 4.14: Single Agent Solution: Rule of OR-Join

- **AND-Fork**. When the control is on an incoming edge of an AND-Fork synchronization bar, then the flow is split into two or more flows of control. The main and unique agent adds all the outgoing edges (i.e., $\{out_i\}$) to the set *active* and removes edges *in* from it. Figure 4.15 illustrates the AND-Fork rule.

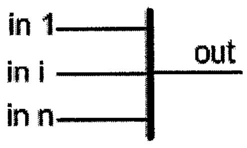
- **AND-Join**. When all incoming edges of an AND-Join are active, their parallel flow must be joined and the control passes to the outgoing edge. The set of edges *active* is updated



```
if CurrConstruct is AND-Fork(in, $\{out_i\}_{i\leq n}$) then
    forall k in $\{out_i\}$
      add k to active
    remove in from active
```

Figure 4.15: Single Agent Solution: Rule of AND-Fork

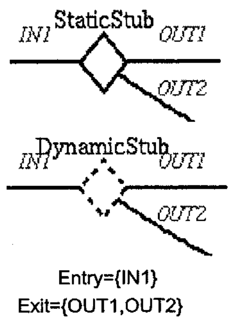
with the addition of edge *out* and the removal of edges in_i from which the control reached the AND-Join. Figure 4.16 illustrates the AND-Join rule.



```
if CurrConstruct is AND-Join($\{in_i\}_{i\leq n}$, out) then
  if active intersect $\{in_i\}$) = $\{in_i\}$ then
    forall k in $\{in_i\}$
      remove k from active
    add out to active
```

Figure 4.16: Single Agent Solution: Rule of AND-Join

- **Stub.** Once the control reaches a stub, the control passes to the selected plug-in and the execution continues following the UCM semantics. The set of edges *active* is updated with the addition of edge *in(startpoint)* of the selected plug-in map and the removal of edges $entry_i$ from which the control reached the stub. Figure 4.17 illustrates the stub rule.



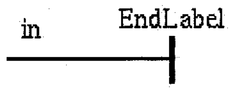
```
if CurrConstruct is Stub($\{entry_i\}_{i\leq n}$,$\{exit_j\}_{j\leq m}$, isDynamic, $[Cond_k]_{k\leq l}$,$[plugin_k]_{k\leq l}$) then
        add (plugin, level) to MapHierarchy
        add (in(StubBinding($entry_i$ , SelectionPolicy( $Cond_k)_{k\leq l}$))) to active
        remove $entry_i$ from active
Where SelectionPolicy:{Cond} $\rightarrow$ $MAPS$ is the selection policy function.
```

Figure 4.17: Single Agent Solution: Rule of Stub

- **End Point.** When the control reaches an end point, four cases have to be considered, depending on whether the end point is connected to a start point (i.e. a waiting place) and whether it is inside a plug-in or part of the root map:

  1. If the end point is connected to a start point(i.e. a waiting place), then the out edge is added to the global list of active edges (i.e. *active*) .
  2. If the end point is inside a plug-in and it is bound to a stub, then the stub's exit point, bound to the plug-in end point, is added to the global list of active edges (i.e. *active*).

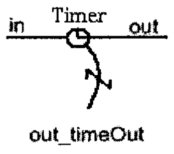
3. If the end point is inside a plug-in but it is not bound to a stub, then the control passes either to any triggered start point part of the plug-in or to the next active edge in the global list of active edges(i.e.*active*) if any.
4. If the end point is part of the root map, then the control passes to the next active edge in the global list of active edges, if any.



```
if CurrConstruct is EndPoint(PostCondition-set, ResultingEvent-set, EndLabel,
in, out) then
if out≠undef then remove in from active
                  add out to active
else
  if UpMap(level)=undef) then remove in from active
  else
    if (StubBinding(EndPoint, level)≠undef)
    then remove in from active
          add out(StubBinding(EndPoint, level)) to active
    else
      if in(triggered(startpoints)≠undef
      then remove in from active
          add in(triggered(startpoints)) to active
      else remove in from active
```

Figure 4.18: Single Agent Solution: Rule of End Point

- **Timer**. The timer rule is very similar to a basic OR-Fork rule with only two disjoint branches (out and out_timeOut).



```
if CurrConstruct is Timer(in, TriggerringEvent-set, out, out_timeout) then
   if (Triggered) then add out to active
   else add out_timeout to active
   else remove in from active
where Triggered: TriggerringEvent-set→Boolean determines whether a trig-
ger occurs within a predefined time frame.
```

Figure 4.19: Single Agent Solution: Rule of Timer

4.3 ASM-UCM Simulation Engine

The ASM-UCM simulation engine is designed for simulating and executing UCM specifications. It is written in AsmL [ASM06] (see Section 2.2.5).

Figure 4.20 shows the architecture of the ASM-UCM simulation engine, which is composed of the following three components: *UCM Specification*, *Data Structures* and *ASM Program*.

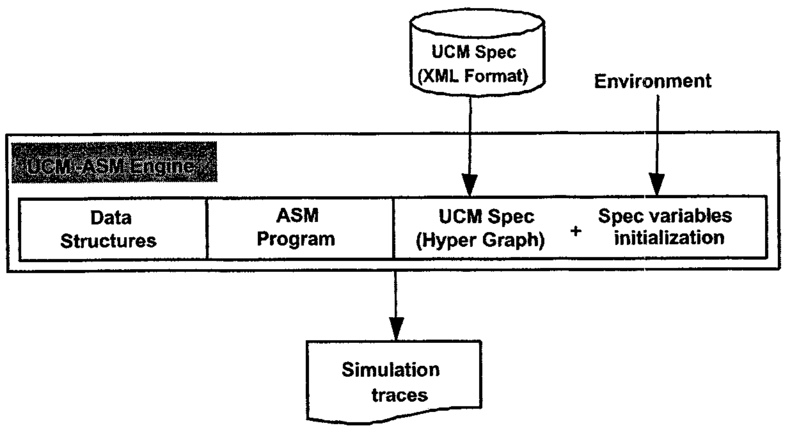


Figure 4.20: ASM-UCM Simulation Engine Architecture

4.3.1 UCM Specification

In order to apply ASM rules defined in Section 4.2.1, the UCM specification (originally described in XML format) should be translated into a hyper graph format according to the syntax defined in Section 4.1.

Note: The translation from the XML format to hyper-graph format is done manually. Before a simulation can be run, the specification's global variables are initialized.

4.3.2 Data Structures

The data structures maintained by the ASM-UCM engine are AsmL structures and dynamic sets. They encode the attribute information of UCM constructs and the structures that handle the dynamic flow of execution. In what follows, we present the data structures that are common to both single-agent and multi-agent solutions. Table 4.1 describes the *UCMConstruct* structure that incorporates many case statements as a way of organizing different variants of UCM constructs. Note that the AsmL set *Hyperedge* denotes the set H of UCM edges.

Table 4.2 describes the following data structures:

- *UCMElement* illustrates the structure of the transition relation λ.
- *Maps* is used to encode the plug-in map details.
- *Stub_Selection* describes the dynamic stub selection policy.
- *OR_Selection* describes the condition-based edge selection of an OR-Fork.
- *Stub_Binding* is used to encode the plug-in/stub binding relation.
- *Mode* is a static universe (where each element is a static nullary function) used to describe the state of an agent.

```
structure UCMConstruct
    case SP_Construct
      in_hy as HyperEdge
      out_hy as HyperEdge
      label as String
      preCondition as BooleanExp
      location as Component
    case R_Construct
      in_hy as HyperEdge
      out_hy as HyperEdge
      label as String
      location as Component
    case EP_Construct
      in_hy as HyperEdge
      out_hy as HyperEdge
      label as String
      postCondition as Boolean
      location as Component
    case OF_Construct
      in_hy as HyperEdge
      Selec as Set of OR_Selection
      label as String
      location as Component
    case OJ_Construct
      in_hy as Set of HyperEdge
      out_hy as HyperEdge
      label as String
      location as Component
```

```
case Stub_Construct
  entry_hy as Set of HyperEdge
  exit_hy as Set of HyperEdge
  Selec_plugin as Set of Stub_Selection
  Binding_Relation as Set of Stub_Binding
  label as String

case AF_Construct
  in_hy as HyperEdge
  out_hy as Set of HyperEdge
  label as String
  location as Component
case AJ_Construct
  in_hy as Set of HyperEdge
  out_hy as HyperEdge
  label as String
  location as Component

case Timer
  in_hy as HyperEdge
  Selec as Set of OR_Selection
  label as String
  location as Component
```

Table 4.1: UCMConstruct Data Structure

```
structure UCMElement
    source as UCMConstruct
    hyper as HyperEdge
    target as UCMConstruct
structure Maps
    label as String
    ele as Set of UCMElement
    ep as Set of EP_Construct
structure Stub_Selection
    stub_plugin as Maps
    stub_cond as BooleanExp
```

```
structure Stub_Binding
    plugin as Maps
    stub_hy as HyperEdge
    start_End as UCMConstruct
structure OR_Selection
    out_hy as HyperEdge
    out_cond as BooleanExp

enum Mode
    running
    inactive
```

Table 4.2: Common Data Structures

Data Structures specific to Multi-Agent Solution and Corresponding Variables

Table 4.3 describes three data structures specific to the multi-agent solution: (1) *MAP_Hierarchy* is used to monitor the run-time transfer of control between stubs and plug-in maps; (2) *STUB_Hierarchy* is used to monitor the run-time hierarchy of traversed stubs during UCM execution;(3) *AJoin_str* is used to monitor the arrival of multiple agents to an AND-Join construct, since an AND-Join is executed only when all its incoming edges are active (by independent agents).

structure MAP_Hierarchy
    current **as** Maps
    up **as** Maps
structure STUB_Hierarchy
    current **as** Stub_Construct
    up **as** Stub_Construct
	structure AJoin_str
	    Ajoin **as** AJ_Construct
	    in_hy **as** HyperEdge

Multi-Agent Specific Global Variables:	

var Map_Hierar **as Set of** MAP_Hierarchy={}
var STUB_Hierar **as Set of** STUB_Hierarchy={}
var AJActive_hyper **as Set of** AJoin_str = {}
var active **as Set of** HyperEdge={}	

Table 4.3: Multi-Agent Data Structures

Data Structures specific to Single-Agent Solution and Corresponding Variables

Table 4.4 describes two data structures specific to the single-agent solution: (1)*SPLUG* is used to handle the run-time hierarchy of both plug-in maps and stubs. (2) *activ* is used to track the set of active edges during the system execution.

structure SPLUG
    st **as** Stub_Construct
    plu **as** Maps
    pl **as Set of** EP_Construct
	structure activ
	    edge **as** HyperEdge
	    level **as** Maps

Single-Agent Specific Global Variables:	

var set_stub_plug **as Set of** SPLUG ={}
var act **as Set of** activ = {}	

Table 4.4: Single-Agent Data Structures

4.3.3 ASM Program

Table 4.5 illustrates the AsmL functions used to access different data structures:

- *GetInHyperEdge* returns the set of incoming edges of a specific UCM construct. It is used in both solutions (i.e., single and multi agents).
- *HyperExists* returns whether an edge is within a set of Edges and it is used to determine which construct to be executed next. This function is common to both solutions.
- *ExecuteResponsibility* executes the chosen responsibility. If the responsibility does not update any variables, *ExecuteResponsibility* simply prints the name of the responsibility. This function is common to both solutions.

- *UpUCMMap* returns the upper map in the run-time map hierarchy. It is used in the multi-agent solution.

- *UpStub* returns the upper stub in the run-time stub hierarchy. It is used in the multi-agent solution.

- *EP_Exists* is used in the ASM rule that corresponds to the case of an end point which is part of a plug-in (i.e., not the rootmap). This function is common to both solutions.

GetInHyperEdge returns the set of incoming edges

```
GetInHyperEdge(i as UCMConstruct) as Set of HyperEdge
  match i
    SP_Construct (a,b,c,d,e): return {a}
    R_Construct (a,b,c,d): return {a}
    EP_Construct (a,b,c,d,e): return {a}
    OF_Construct (a,b,c,d): return {a}
    Stub_Construct(a,b,c,d,e): return a
    AF_Construct (a,b,c,d): return {a}
    AJ_Construct (a,b,c,d): return a
    OJ_Construct (a,b,c,d): return a
    TM_Construct (a,b,c,d): return a
```

HyperExists returns whether an edge is within a set of Edges

```
HyperExists(i as HyperEdge, j as Set of HyperEdge) as Boolean
   return (exists k in j where k=i)
```

ExecuteResponsibility executes the chosen responsibility

```
ExecuteResponsibility(R as R_Construct)
  WriteLine("Responsibility:" + R.label + " in component:" + R.location)
  if (R = R1) then
    Var1.value := true
  if (R = R2) then
    Var2.value := false
...
```

UpUCMMap returns the upper map in the run-time map hierarchy

```
UpUCMMap(i as Maps) as Maps
   choose v in Map_Hierar where v.current = i
     return v.up
```

UpStub returns the upper stub in the run-time stub hierarchy

```
UpStub(i as Stub_Construct) as Stub_Construct
    choose v1 in STUB_Hierar where v1.current = i
      return v1.up
    ifnone return i
```

EP_Exists returns whether an end point is part of a set of end points

```
EP_Exists(i as EP_Construct, j as Set of EP_Construct) as Boolean
    return (exists k in j where k=i)
```

Table 4.5: AsmL Access Functions

Multiple Agent Solution Figure 4.21 illustrates the class *Agent*, a sketch of the ASM rules and the main program of the ASM-UCM simulation engine.

Single Agent Solution Figure 4.22 illustrates a sketch of the ASM rules and the main program of the single agent solution.

```
class Agent
   const id as String
   var active as Edge
   var mode as Mode
   var level as Maps
Program()
step
    until me.mode = inactive
    do
     choose h in level.ele where HyperExists(active, GetInEdge(h.source))
     match (h.source)
// Rule of Start Point
     SP_Construct (a,b,c,d): step
        if d.Value() = true
         me.active := b
        else
         WriteLine("Start Point:" + c + "Check the preconditions")
         me.mode := inactive
// Rule of Responsibility
     R_Construct (a,b,c): ExecuteResponsibility(h.source as R_Construct)
                          me.active := b
// Rule of OR-Fork
      OF_Construct (a,b,c,d): step
        choose v in b where (v.out_cond).Value() = true
          me.active := v.out_hy
        ifnone
         me.mode := inactive
// ...
Main()
   var todo = StartPoints
   step while todo.Count > 0
    choose a in todo
    todo(a) := false
    let ag = new Agent("Tel System:", a.in_hy, running, RootMap, init_stub)
    ag.Program()
```

Figure 4.21: Multi-Agent Solution: ASM-UCM program

```
class Agent
   const id as String
   var active as Edge
   var mode as Mode
Program()
step
  until ((act = {}) or (me.mode = inactive))
  do
   let h = {t1.edge || t1 in act }
   choose z in act
    choose h in level.ele where HyperExists(active, GetInEdge(h.source))
     match (s2.source)
// Rule of Start Point
     SP_Construct (a,b,c,d): step
         if d.Value() = true
          add activ(b, z.level) to act
          choose r in act where r.edge = a
          remove r from act
         else
          WriteLine("Start Point:" + c + "Check the preconditions")
          me.mode := inactive
// Rule of Responsibility
      R_Construct (a,b,c): ExecuteResponsibility((s2.source) as R_Construct)
         add activ(b, z.level) to act
         choose r in act where r.edge = a
         remove r from act
// Rule of OR-Fork
       OF_Construct (a,b,c,d): step
        choose v in b where (v.out_cond).Value() = true
         add activ(b, z.level) to act
         choose r in act where r.edge = a
         remove r from act
        ifnone
         WriteLine("Please check conditions of OR-Fork:" + c)
         me.mode := inactive
// ...
Main()
   step
    forall i in StartPoints
     add activ(GetInHyperEdge(i), RootMap) to act
   step
     choose a in StartPoints
     let ag = new Agent("Tel System:", a.in_hy, running, RootMap, init_stub)
     ag.Program()
```

Figure 4.22: Single-Agent Solution: ASM-UCM program

4.4 General Discussion

4.4.1 Interpretation vs. Compilation

Our ASM-UCM simulation engine is based on the *interpretation* concept of execution. It looks at each element of the UCM specification, works out what it means, executes its corresponding rule and then goes onto the next UCM element, while the approach proposed in [Amy94] is based on a compilation concept. Indeed, the modification of the semantics of a UCM construct will result in changing the corresponding ASM rule without modifying the original specification. However in [Amy94], one needs to redesign the mapping between UCM to LOTOS and to regenerate the LOTOS specification.

4.4.2 Language Evolution

Our ASM rules can be easily modified to accommodate language evolution. Indeed, the modification of the semantics of a UCM construct or the addition of a new construct result in the modification or the addition of a new ASM rule that describes the semantics of the new construct.

4.4.3 Semantic Variations

The proposed ASM-UCM simulation engine may support different semantic variations at minimal cost. In the context of concurrency models, agents may behave either in interleaving semantics with atomic actions (i.e. comparable to LOTOS processes [ISO89]) or in true concurrency mode. The choice of the suitable alternative depends on the application domain and the ASM program (i.e., ASM Scheduler) is designed accordingly.

4.4.4 Extraction of Information

The ASM-UCM simulation engine can be instrumented to capture all aspects of UCM specification. This includes the name and the type of each executed construct, values of variables of interest at each computation step, names of traversed stubs and plug-in maps, and component names. In Chapter 7, we will introduce the notion of time into UCMs and will show how an ASM-based semantics will help capture the specification temporal aspect as well.

4.5 AsmL Specification of the Simple Telephony System

Table 4.6 shows two abstract types:(1) Edge that contains all the edges of the specification and (2) Component that contain all the components of the specification. Four Boolean variables are defined as well. For instance, we consider that the originating user is subscribed to OCS (i.e., *subOCS* := true) and the terminating user is subscribed to CND (*subCND* := true).

Figures 4.23, 4.24, 4.25, 4.26 and 4.27 illustrate respectively the AsmL implementation of the default plug-in map, the OCS plug-in map, the stub *SOrig*, the stub *Sterm* and the root map.

enum Hyperedge	**enum** Component
e1	UserOrig
e2	UserTerm
e3	AgentOrig
e4	AgentTerm
...	Unbound
Global Variables:	
subOCS = **new** BooleanValue(true)	
subCND = **new** BooleanValue(true)	
OnList = **new** BooleanValue(true)	
busy = **new** BooleanValue(true)	

Table 4.6: Edges, Components and Global Variables

```
var DEF_start as SP_Construct=SP_Construct(DEF_in1, DEF1 ,"Start", BooleanVar(pre_cond_start),
AgentTerm)
var DEF_continue as EP_Construct=EP_Construct(DEF1, h0, "continue", true, AgentTerm)
var DEF_Plugin as Maps=Maps("DEF_Plugin", {UCMElement(DEF_start, DEF1, DEF_continue),
UCMElement (DEF_continue, h0, DEF_continue)}, {DEF_continue})
```

Figure 4.23: AsmL Default Plug-in Map

```
var OCS_start as SP_Construct=SP_Construct(OCS_in1, OCS1, "Start", BooleanVar(pre_cond_start),
AgentOrig)
var checkOCS as R_Construct=R_Construct(OCS1, OCS2, "checkOCS", AgentOrig)
var OCS_OF1 as OF_Construct=OF_Construct(OCS2, {OR_Selection(OCS3, -BooleanVar(OnList)),
OR_Selection(OCS4, BooleanVar(OnList))}, "OCS_OF1", AgentOrig)
var deny as R_Construct=R_Construct(OCS4, OCS5, "deny", AgentOrig)
var OCS_fail as EP_Construct=EP_Construct(OCS5, h0, "fail", true, AgentOrig)
var OCS_success as EP_Construct=EP_Construct(OCS3, h0, "success", true, AgentOrig)
var OCS_Plugin as Maps=Maps("OCS_plugin",{UCMElement(OCS_start, OCS1, checkOCS),
UCMElement(checkOCS, OCS2,OCS_OF1), UCMElement(OCS_OF1, OCS3, OCS_success), UCMEle-
ment(OCS_OF1, OCS4, deny), UCMElement(deny, OCS5, OCS_fail), UCMElement(OCS_fail, h0,
OCS_fail), UCMElement(OCS_success, h0, OCS_success)}, {OCS_success, OCS_fail})
```

Figure 4.24: AsmL: OCS Plug-in Map

```
var Orig_start as SP_Construct=SP_Construct(Orig_in1, O1, "Start", BooleanVar(pre_cond_start), Agen-
tOrig)
var snd_req as R_Construct =R_Construct(O2, O3, "snd_req ", AgentOrig)
var Orig_fail as EP_Construct=EP_Construct(O4, h0, "fail", true, AgentOrig)
var Orig_success as EP_Construct=EP_Construct(O3, h0, "success", true, AgentOrig)
var Sscreen as Stub_Construct=Stub_Construct({O1},{O2, O4}, {Stub_Selection (OCS_Plugin, Boolean-
Var(subOCS)), Stub_Selection (DEF_Plugin,-BooleanVar(subOCS))}, {Stub_Binding(OCS_Plugin, O1,
OCS_start), Stub_Binding(OCS_Plugin, O2, OCS_success),Stub_Binding(OCS_Plugin, O4, OCS_fail),
Stub_Binding(DEF_Plugin, O1, DEF_start),Stub_Binding(DEF_Plugin,O2,DEF_continue)}, "Sscreen")
var Orig_Plugin as Maps =Maps("Orig_plugin", {UCMElement(Orig_start, O1, Sscreen), UCMEle-
ment(Sscreen, O2, snd_req), UCMElement(Sscreen, O4, Orig_fail), UCMElement(snd_req, O3, Orig_success),
UCMElement(Orig_success, h0, Orig_success), UCMElement(Orig_fail, h0, Orig_fail)},{Orig_success,
Orig_fail})
var Sorig as Stub_Construct=Stub_Construct({e1},{e2,e4}, {Stub_Selection(Orig_Plugin, Boolean-
Var(_true))}, {Stub_Binding(Orig_Plugin, e1, Orig_start), Stub_Binding(Orig_Plugin, e2, Orig_success),
Stub_Binding(Orig_Plugin,e4, Orig_fail)}, "SOrig")
```

Figure 4.25: AsmL: Stub SOrig

```
var term_start as SP_Construct=SP_Construct (term_in1 , T1, "Start", BooleanVar(pre_cond_start), Agent-
Term)
var term_OF1 as OF_Construct=OF_Construct(T1, {OR_Selection(T2, -BooleanVar(busy)),
OR_Selection(T3, BooleanVar(busy))}, " term_OF1", AgentTerm)
var term_AF1 as AF_Construct=AF_Construct(T2 ,{T5,T9}, "term_AF1", AgentTerm)
var ringTreatment as R_Construct=R_Construct(T6, T7, "ringTreatment", AgentTerm)
var ringingTreatment as R_Construct=R_Construct(T9, T10, "ringingTreatment", AgentTerm)
var reportSuccess as EP_Construct=EP_Construct(T10, h0 , "reportSuccess", true, AgentTerm)
var busyTreatment as R_Construct=R_Construct(T3, T4, " busyTreatment ", AgentTerm)
var term_fail as EP_Construct=EP_Construct(T4, h0 , "fail", true, AgentTerm)
var term_success as EP_Construct=EP_Construct(T7, h0 , "success", true, AgentTerm)
var term_disp as EP_Construct=EP_Construct(T8, h0 , "display", true, AgentTerm)
var term_Plugin as Maps=Maps("term_Plugin",{UCMElement(term_start, T1, term_OF1),
UCMElement(term_OF1, T2, term_AF1), UCMElement(term_OF1,T3 , busyTreat-
ment), UCMElement(busyTreatment, T4, term_fail), UCMElement(term_AF1, T5, Sdis-
play),UCMElement(Sdisplay,T6,ringTreatment), UCMElement(ringTreatment,T7, term_success),
UCMElement(Sdisplay, T8, term_disp), UCMElement(term_AF1, T9, ringingTreatment), UCMEle-
ment(ringingTreatment, T10, reportSuccess), UCMElement(term_disp, h0, term_disp), UCMEle-
ment(reportSuccess, h0, reportSuccess), UCMElement(term_fail, h0, term_fail), UCMElement(term_success,
h0, term_success)}, { reportSuccess, term_fail, term_success, term_disp})
var Sterm as Stub_Construct=Stub_Construct({e2},{e3, e5, e7, e9},{Stub_Selection(term_Plugin, Boolean-
Var(_true))}, {Stub_Binding(term_Plugin, e2, term_start), Stub_Binding (term_Plugin, e3, term_success),
Stub_Binding(term_Plugin, e9, term_disp), Stub_Binding (term_Plugin, e5, term_fail),Stub_Binding
(term_Plugin, e7, reportSuccess)}, "Sterm")
```

Figure 4.26: AsmL: Stub Sterm

```
var root_req as SP_Construct=SP_Construct (in1, e1, "Req", BooleanVar(pre_cond_start), UserOrig)
var root_fwd_sig1 as R_Construct=R_Construct(e5 , e6, "fwd_sig",AgentOrig)
var root_fwd_sig2 as R_Construct=R_Construct(e7 , e8, "fwd_sig", AgentOrig)
var root_ring as EP_Construct=EP_Construct(e3 , h0, "ring", true, UserTerm)
var root_display as EP_Construct=EP_Construct(e9 , h0, "display", true, UserTerm)
var root_notify as EP_Construct=EP_Construct(e4 , h0, "notify", true, UserOrig)
var root_busy as EP_Construct=EP_Construct(e6 , h0, "busy", true, UserOrig)
var root_ringing as EP_Construct=EP_Construct(e8 , h0, " ringing", true, UserOrig)
var RootMap as Maps = Maps("RootMap", {UCMElement(root_req , e1, Sorig), UCMElement(Sorig , e4,
root_notify), UCMElement(Sorig , e2, Sterm), UCMElement(Sterm , e3 , root_ring), UCMElement(Sterm,
e9, root_display), UCMElement(Sterm, e5, root_fwd_sig1), UCMElement(Sterm, e7, root_fwd_sig2),
UCMElement(root_fwd_sig1, e6, root_busy), UCMElement(root_fwd_sig2, e8, root_ringing), UCMEle-
ment(root_ring, h0 ,root_ring), UCMElement(root_display, h0, root_display), UCMElement(root_ringing,
h0, root_ringing), UCMElement(root_busy, h0, root_busy), UCMElement(root_notify, h0, root_notify)},
{root_ring, root_display, root_ringing, root_notify})
```

Figure 4.27: AsmL implementation of the Root Map

4.6 Chapter Summary

In this chapter, we have presented a formal syntax and a formal operational semantics for Use Case Maps language based on Abstract State Machines. Our ASM models provide a concise semantics of UCM functional constructs and describes precisely the control semantics. Two possible ASM-based solutions were presented:

- Multi agent ASM solution: Each sequential UCM segment is represented by an independent agent that runs to completion before a new one can start executing. Each agent maintains a single active edge at every execution point.

- Single agent ASM solution with non deterministic interleaving. This solution considers a unique thread of execution and a unique set of active edges (i.e., a.active is a set of edges). Contrary to the multi-agent solution, the single-agent solution ensures a full non-deterministic interleaving between concurrent threads.

Both solutions are implemented within the ASM-UCM simulation engine (see Section 4.3), designed for simulating and executing UCM specifications. Finally, Section 4.5 presents the AsmL specification of the Simple Telephony System.

Chapter 5

Early Stages Validation Approach

As requirement descriptions evolve, they quickly become error-prone and difficult to understand. Errors in a requirements model have prolonged detrimental effects on reliability, cost, and safety of a software system. It is very costly to fix these errors in later phases of software development if they cannot be corrected during requirements analysis and design [SL03]. Thus, the development of techniques and tools to support requirement specification development, understanding, testing, maintenance and reuse becomes an important issue.

Among the rigorous validation techniques, testing and simulation are the most powerful methods because the behavior of a system can be tested and observed. Execution is a powerful and direct mechanism to observe a system and its behavior. When practitioners execute a system and, find some unexpected results or their understanding of the system requirements was wrong, it usually means that errors exist in a system.

In this chapter[1], we combine well-known techniques (i.e., slicing, step by step simulation and trace generation) to validate requirement specifications described using the Use Case Maps notation. In the next section, an overview of the validation approach is described. In Section 5.2, we extend the well-known source code analysis technique *program slicing* [Wei84, KL88] to functional requirement specifications, based on the UCM notation. This new application of slicing, called *UCM Requirement Slicing* helps reduce the complexity of the requirement specifications and facilitates requirement comprehension and maintenance.

5.1 High Level Validation Approach

Figure 5.1 illustrates the proposed validation approach. The approach is iterative and composed of four iterations:(1) Instrumenting the execution environment; (2) Simulation and generation of traces; (3) Inspection of traces ; (4) Fixing errors.

[1]This chapter content is published in System Analysis and Modeling- SAM 2004 [HDR04], in IEEE International Workshop on Principles of Software Evolution - IWPSE 2005) [HJRD05] and in IEEE International Workshop on Principles of Software Evolution - IWPSE 2007) [SHR07]

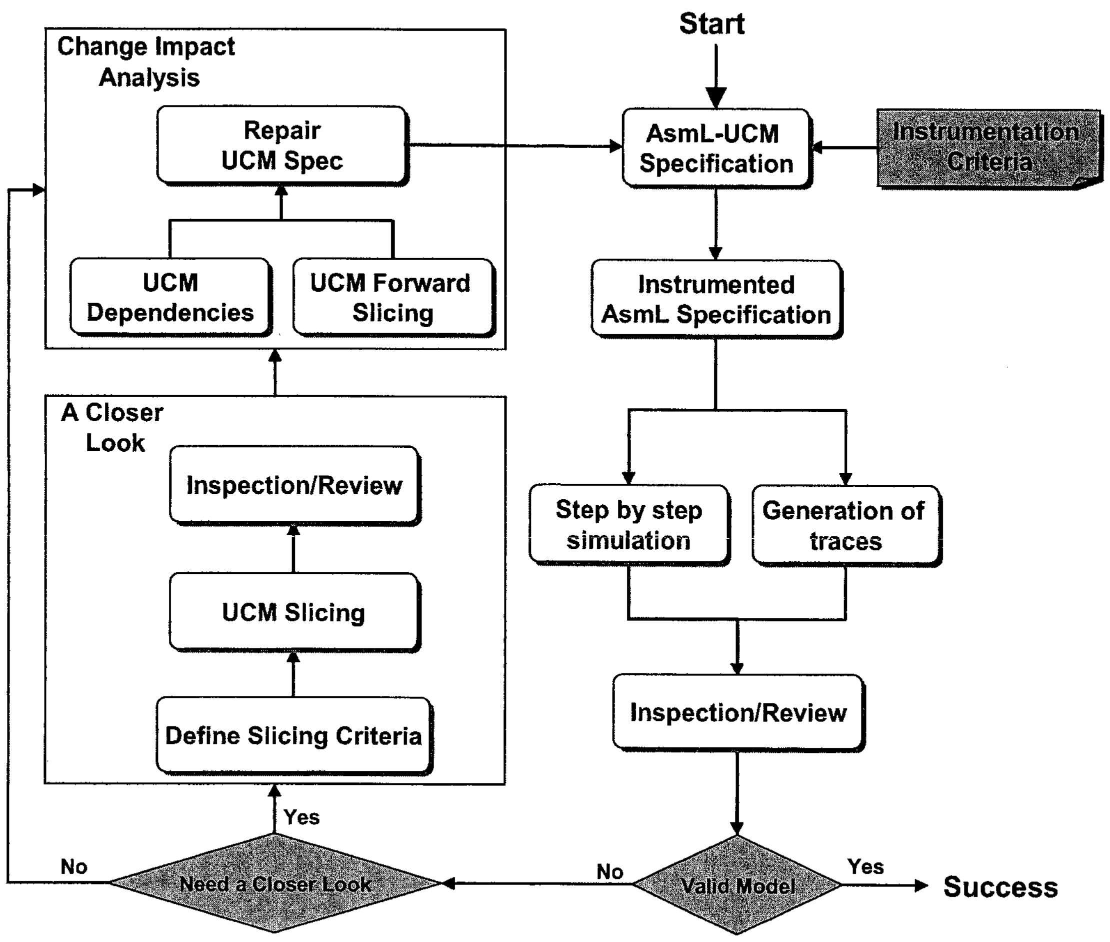


Figure 5.1: Early Stages Validation Approach

5.1.1 Instrumenting the ASM-UCM simulation engine

One technique for collecting programs run-time information consists of instrumenting the execution environment in which the system runs. The ASM-UCM simulator can be instrumented to collect information of interest such as:

- Executed UCM constructs.
- Values of selected global variables after the execution of specific types of constructs.
- Executed stubs and plug-in maps.
- UCM components.

The simulator can also be instrumented to record how many times a given construct or plug-in has been exercised in each simulation trace. The coverage data collected from the traces can be used to analyze the design.

5.1.2 Simulation and Generation of Traces

The use of traces or execution histories as an aid to debugging, testing and analysis is a well established technique for programming languages. It is very common that traces, once generated, are saved in text files. In the context of UCM specifications, a trace file starts with a start point of the root map and terminates with one end point of the root map (in between, we can have additional start points and end points executions). A trace is composed of a sequence of lines. Each line records the name of the UCM construct and its location (i.e. UCM component). Values of variables of interest can also be printed at each computation step.

5.1.3 Inspection of Traces

The resulting traces are then inspected (by an analyst). At the UCM abstraction level the following design errors can be discovered:

- Lock situations: At the Use Case Maps level of abstraction, no distinction is made between deadlock, livelock and other liveness error situations. A UCM trace should usually terminate with the execution of an end point that is part of the root map. A lock situation is detected when the trace does not terminate with an end point that belongs to the root map. Such lock may be due to wrong conditions at a branching construct or plug-in selection policy.
- Violations of user specified correctness assertions: in the context of UCMs, concurrency and non-determinism may impact causality assertions. For instance, analysts may use a generated trace to check for assertions of the following form: *responsibility R2 should always be preceded by responsibility R1.* Furthermore, invariants can be checked by parsing the values of variables at each computation step and computing the invariant expression.

- Violations of postconditions: trace-based testing is efficient to validate that a postcondition holds for every execution of a responsibility that satisfies a precondition. To validate postconditions we need to accomplish the following steps:(1) Identify the execution of the responsibility that is a target for postcondition validation (2)ensure the precondition and (3)validate the postcondition.

- Unreachable specification parts: in computer programming, unreachable code, or dead code, is code that exists in the source code of a program but can never be executed. In the context of UCMs, unreachable specification parts may be detected, for instance, by exercising all possible combinations of the values of global Boolean variables and with all combinations of sequences of start points.

Note: Replaying execution traces and measuring how thoroughly a specification has been exercised are out of the scope of this thesis.

5.1.4 Fixing Errors and/or Design Improvement

For a number of reasons, the number and size of generated traces can be too large making the specification difficult to validate. In our research, we address these issues by introducing a new approach to reduce the complexity of the requirement specifications. Our new approach is based on *slicing techniques* to guide requirement engineers, designers and programmers during the comprehension and analysis process of requirement specifications.

A large UCM specification may be reduced to reflect only parts that are relevant to some specific criteria. The obtained slice is then analyzed (through simulation and trace generation) and potential errors can be fixed. UCM slices help analyze to what extent the behavior and/or architecture of the system might be affected by a specific validation/maintenance task. For each slice, the analyst/maintainer can identify the part of the particular scenario that contributes to the slicing criterion (on both architectural and behavioral parts). Therefore, step by step execution can be executed once we have a reduced specification. Furthermore, fixing errors may introduce new collateral errors. We propose a UCM-based *Change Impact Analysis* technique to assess the impact of a change in order to minimize the probability of introducing new errors.

5.2 Slicing Use Case Maps Requirement Specifications

5.2.1 Traditional Program Slicing

Program slicing was originally introduced as a technique to simplify programs to provide support during debugging and program comprehension [Tip95, Wei84] and has been applied to a wide variety of problems including: program understanding, maintenance [GL91], debugging, differencing, integration and testing [Tip95]. Program slicing, a program reduction technique, allows one to reduce the size of the source code of interest by identifying only those parts of the original program that are relevant to the computation of a particular function/output of interest [Wei84]. Moreover, slicing preserves the program semantics of the original program with respect to the slicing criterion [Wei84].

The notion of program slicing originated in the seminal paper by Weiser [Wei84]. Weiser defined a slice S as a reduced, executable program P′ obtained from a program P by removing statements such that S replicates parts of the behavior of the program. Informally, a static program slice consists of those parts of a program that potentially could affect the value of a variable V at a point of interest. The resulting slice shown in Table 5.1 indicates which statements influence the output of the variable *sum* at line 12.

	Original Program	Resulting Slice
1	begin	begin
2	read(n);	read(n);
3	i:=1;	i:=1;
4	sum:=0;	sum:=0;
5	pro:=1;	
6	while (i≤n)	while (i≤n)
7	begin	begin
8	sum:=sum+i;	sum:=sum+i;
9	pro:=pro*i;	
10	i:=i+1;	i:=i+1;
11	end;	end;
12	write(sum);	write(sum);
13	write(pro);	
14	end	end

Table 5.1: Example of Program Slicing

Backward Slicing vs. Forward Slicing. Having picked a slicing criterion one of two forms of slice can be constructed: a backward slice or a forward slice. The former consists of all the statements of the program that affect a given point in the program (i.e. the slicing criterion), whereas a forward slice contains those statements of the program which are affected by the slicing criterion. Backward slices can assist a developer by helping to locate the parts of the program which contain a bug. Forward slicing can be used to predict the parts of a program that will be affected by a modification.

Dynamic Slicing vs. Static Slicing. Korel and Laski introduced in [KL88] the notion of dynamic slicing that can be seen as a refinement of the static approach. The dynamic slice preserves the program behavior for a specific input, in contrast to the static approach, which preserves the program behavior for the set of all inputs for which a program terminates. Dynamic program slicing may significantly reduce the size of a program slice because run-time information, collected during program execution, is used to compute program slices. Dynamic program slicing was originally proposed only for program debugging, but its application has been extended to program comprehension, software testing, and software maintenance. Different types of dynamic program slices, together with algorithms to compute them, have been proposed in the literature.

Chopping. A related operation is program chopping [JR94, RR95]. A chop consists of all program points affected between one point (the chop source) and another (the chop target). A chop answers questions of the form: *which program elements serve to transmit effects from a given source element S to a given target T?*. In the example of Table 5.1, the chop between the initialization of variable *sum* and the output of variable *i* is empty, reflecting the fact that there is no information flow between the source and the target.

5.2.2 Related Work: Model Based Slicing

Different slicing techniques and criteria are required because various applications require different properties of slices. In recent years, the application of slicing has been extended to other software artifacts [SH96] including: software architecture[Zha98], requirement models [HW97, KSTV03] and formal specification [CR94, MT98]. A detailed survey of different slicing techniques and their applications can be found in [Luc01, Tip95, LRG04].

Slicing of Hierarchical State Machines. Heimdahl et al. [HW97] apply slicing to the requirement specification language RSML (Requirement State Machine Language). Their proposed method consists on reducing the requirement specification based on a specific scenario of interest. The reduced specification contains only the behaviors that are possible when the operating conditions defining the reduction scenario are satisfied. Such a reduced specification is called the *interpretation* of the specification under this scenario. Next, the produced interpretation is sliced based on different entities in the model to highlight the portions of the specification affecting an output variable or a specific transition. This is achieved through a data and control flow information analysis. The slices can be arbitrarily combined using standard set of operations to construct a combined slice containing the information of interests.

Slicing of State Based Models. Korel et al. [KSTV03] presented an approach of slicing EFSM (Extended Finite State Machines) models. Their approach produces an EFSM slice based on EFSM dependence analysis. The resulting slice may further be reduced by merging states and transitions to construct a non-deterministic EFSM. This is called non-deterministic slicing.

RSML and EFSM slicing emphasizes only the behavioral part of the requirement specification. The architectural part is left aside. Use Case Maps scenarios combine both aspects (i.e. behavioral and architectural) in a single representation. Our proposed technique takes advantage of this dual representation.

Architectural Slicing. Zhao [Zha98] introduced a new form of slicing called *Architectural slicing* to aid architectural understanding and reuse. He applied slicing to an architectural specification of a software system written in *WRIGHT*, which is an *Architectural Description Language* (ADL). A *WRIGHT* architectural specification of a system is defined by a set of component and connector type definitions, a set of instantiations of specific objects of these types, and a set of attachments. Attachments specify which components are linked to which connectors. Each component has an interface defined by a set of ports and each connector has an interface defined by a set of roles. In order to compute an architectural slice, an architecture information flow graph is constructed then a traversal algorithm is applied. The reduced architectural description contains only the lines of ADL code that could be associated with a particular slicing criterion. In [SW98, Zha98] the slicing criterion is either a set of ports of a component or a set of roles of a connector. Stafford et al. [SW98] presented a closely related method to Zhao's work. They introduced a software architecture dependency technique called chaining. Their work consists on extracting a chain of dependences (called *links*) between the specification's elements based on a set of ports of a component (Slicing criterion).

5.2.3 UCM Slices

In contrast to traditional program slicing, requirement slicing is designed to operate on the requirement specification of a system, rather than the source code of a program. The resulting requirement slice provides knowledge about high-level structure of a system, rather than its low-level implementation details.

Our work on UCM slicing builds on from prior work in the following two primary areas: functional requirement slicing and architectural slicing. Intuitively, a UCM slice may be viewed as a subset of the behavior of a global UCM. While a traditional slice intends to isolate the behavior of a specified set of program variables, a UCM slice intends to isolate a set of scenarios that lead to a specific behavior.

When a UCM slicer is invoked, it takes as input:

1. A complete system requirement specification based on the UCM notation.
2. A slicing criterion.

Depending on the user's interest, the UCM slicer computes a backward or forward slice with respect to the selected slicing criterion. In order to define a *UCM slice*, we introduce the concepts of reduced UCM elements.

Definition 2 (Reduced UCM elements) *Let $RS = (D, H, \lambda, C, GVar, B_c, B_s)$ be an UCM Requirement Specification(see Definition 1).*

- *A reduced domain is a set D' that is derived from D by removing zero, or more elements (i.e. $D' \subseteq D$).*
- *A reduced set of edges is a set H' that is derived from H by removing zero, or more elements (i.e. $H' \subseteq H$).*
- *A reduced transition relation λ' is a relation derived from λ by removing zero or more tuples.*
- *A reduced component c' is a component that has less functionalities than the original component.*
- *A reduced guard set $GVar'$ is a set $GVar' \subseteq GVar$ that is derived from $GVar$ by removing zero, or more expressions.*

- *A reduced component binding relation $B_c{}'$ is a relation derived from B_c by removing zero or more couples.*
- *A reduced stub binding relation $B_s{}'$ is a relation derived from B_s by removing zero or more tuples.*
- *A reduced stub is a stub that contains reduced plug-in maps and may have fewer plug-in maps than the original stub.*

Given a UCM, our goal is to compute a UCM slice which corresponds to a subset of the original UCM that preserves the semantics of the UCM with respect to chosen slicing criterion.

Note: We can have as a result a set of flat scenarios (i.e. sequential traces where neither concurrencies nor choices are involved). However the original UCM semantics will not be preserved.

Definition 3 (Reduced UCM) *Let $RS = (D, H, \lambda, C, GVar, B_c, B_s)$ and $RS' = (D', H', \lambda', C', GVar', B_c{}', B_s{}')$ be two UCM requirement specification. RS' is a reduced specification of RS if:*

- *D' is a reduced set of D.*
- *H' is a reduced set of H.*
- *λ' is a reduced transition relation of λ*
- *$C' = c_1{}', c_2{}', \ldots, c_n{}'$ is a subset of C such that for k=1,2,...,n. $c_k{}'$ is a reduced component of c_k.*
- *$GVar'$ is a reduced set of GVar.*
- *$B_c{}'$ is a reduced component binding relation of B_c.*
- *$B_s{}'$ is a reduced stub binding relation of B_s.*

Note: Since a plug-in is also a stand alone UCM, a reduced plug-in can be defined in the same way as a reduced UCM.

5.2.4 UCM Slicing Criteria

The selection of a slicing criterion depends on the particular analysis task. The focus is frequently on examining the requirement with respect to a particular system functionality, e.g a particular system feature or a particular behavior.

Based on the task and the degree of system understanding a user may choose between specifying (1) a UCM construct as a slicing criterion (regardless of its location), or (2) a UCM component to focus the analysis on one specific component, or (3) a construct and a specific component to focus the analysis on one construct specific to one component).

Definition 4 (UCM Slicing criteria) *Let RS be a UCM requirement specification. A slicing criterion (SC) for RS may be:*

- *A UCM construct (e.g., responsibility, start/end point, etc.). If the chosen UCM construct is not part of the root map, the stub and the plug-in to which it belongs must be defined as well.*
- *A UCM component.*
- *A combination of a UCM construct and a UCM component.*

Note: For bound UCM specifications, the computed slice is also bound even when the slicing criteria do not involve a specific component.

5.2.5 Slicing UCM Constructs

Figure 5.2 shows different UCM constructs and their potential reduced versions after applying backward slicing. E is a generic end point which is added after the SC to form a valid reduced UCM. In the reduced OR-Fork (Figure 5.2(aa)), only one path is included in the reduced UCM. In the reduced OR-Join (Figure 5.2(bb)), the non-determinism is preserved. In the reduced AND-Fork (Figure 5.2(cc)), the interleaving semantics is preserved, since concurrent responsibilities SC and d may occur in different order($SC;d$ or $d;SC$).

Figure 5.2(gg) shows the slice obtained for a UCM with a dynamic stub. The selection policy between plug-in 1 and plug-in 2 is based on the value of global variable C:(1) (C=*true*) $\rightarrow$ *Plug-in 1* (connects IN1 to OUT1)(2) (C=*false*) $\rightarrow$ *Plug-in 2* (connects IN1 to OUT2). *Plug-in 1* is sliced out because its end point is bound to end point E2. The resulting stub is a reduced stub with only one exit point *E1* containing *Plug-in 2*.

5.2.6 UCM Backward Slicing

In what follows, we present our UCM backward slicing algorithm, which is based on a backward traversal of the UCM specification. While performing the backward traversal, the slicer collects all the logical predicates, defined on UCM global variables, leading to the execution of the targeted criterion and produces what we refer to as *reachability expression*. The reachability expression is solved by finding the initial variable values and/or the sequence of inputs that the environment has to provide to be able to reach the slicing criterion. Table 5.2 describes the high level schema of the UCM slicing algorithm.

Logical conditions are collected as the traversal progresses. Each stub defines a level of abstraction and is treated separately. Therefore, we obtain reduced stubs at different abstraction levels. Since a plug-in can be installed in many stubs according to the chosen scenario, the targeted stub and plug-in to which the SC belongs. This information is essential because of the '*Many-Many*' association between plug-in maps and stubs.

It should be noted that the presented algorithm is not necessarily the most time and space efficient approach to compute UCM slices. The algorithm will terminate due to the backward traversal step and the fact that there is a finite number of responsibilities in the UCM.

UCM slices based on the other two types of slicing criteria are obtained as follows:

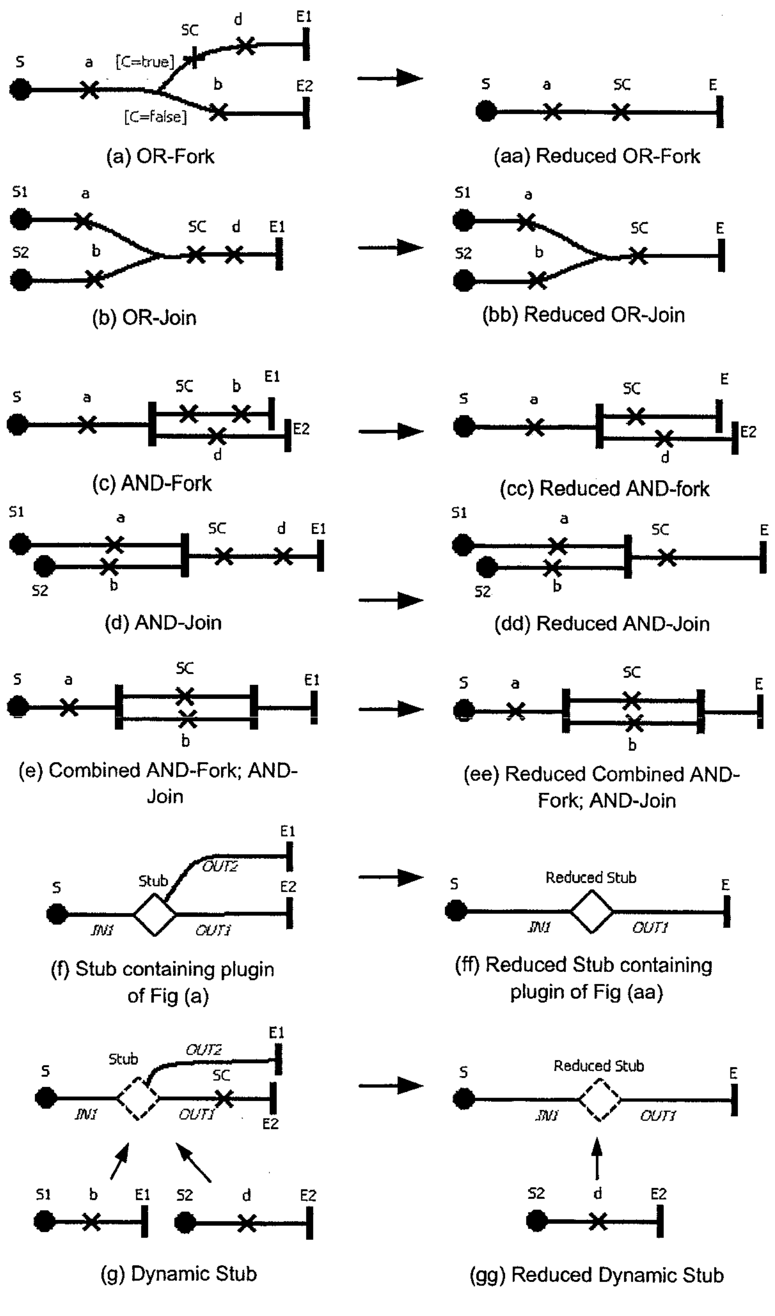


Figure 5.2: UCM Constructs and its Reduced Form

```
Input:UCM Spec + (Slicing criterion SC, level, stub)
Output: Reduced UCM Spec, Reachability Expression
```

```
Step1:(* Searching SC *)
while (target(elem) ≠ SC) do /* elem of type UCMElement */
        elem := next(elem) /* next is defined such that target(elem)=source(next(elem)) */
end while
if (level=RootMap) then
      ReducedSlice := elem ∪ (SC,e,E)
      goto step2
else
      ReachabilityExp := SelectionCond(Stub) /* Selection policy for stubs */
      ReducedPlugin := el ∪ (SC,e,E)
end if
Step2:(* UCM Backward Traversal + collect conditions *)
if (level=RootMap) then
  while (type(source(elem))≠ StartPoint) do
    if (type(source(elem))= OF) then ReachabilityExp := ReachabilityExp ∧ Cond(source(elem))
          ReducedSlice:=ReducedSlice ∪ elem
    if (type(source(elem))= OJ) then ReducedSlice:=ReducedSlice ∪ elem
    if (type(source(elem))= AF) then
      ReducedSlice:=ReducedSlice ∪ elem
      ele:=elem
     while (type(target(ele)) ≠ AJ) ∧ (type(target(ele)) ≠ EP) do
       ReducedSlice:=ReducedSlice ∪ ele
       ele := next(ele)
    if (type(source(elem))= AND-Join) then ReducedSlice:=ReducedSlice ∪ elem
    if (type(source(elem))= Stub) then
     ReducedSlice := ReducedSlice ∪ elem
     Select only Stub'plugins bound to the exit point of source(elem)
     ReducedSlice := ReducedSlice ∪ elem
    elem := previous(elem) /* previous is defined such that
source(elem)=target(previous(element)) */
  end while
else
  while (type(source(elem))≠ StartPoint) do
    if (type(source(elem))= OF) then
     ReachabilityExp := ReachabilityExp ∧ Cond(source(elem))
     ReducedPlugin:=ReducedSlice ∪ elem
    if (type(source(elem))= OJ) then ReducedPlugin:=ReducedPlugin ∪ elem
    if (type(source(elem))= AF) then
     ReducedPlugin:=ReducedPlugin ∪ elem
      ele:=elem
     while (type(target(ele)) ≠ AJ) ∧ (type(target(ele)) ≠ EP) do
       ele := next(ele)
       ReducedPlugin:=ReducedPlugin ∪ elem
     end while
    if (type(source(elem))= AND-Join) then ReducedPlugin :=ReducedPlugin ∪ elem
    if (type(source(elem))= Stub) then
     ReducedPlugin := ReducedPlugin ∪ elem
     Select only Stub'plugins bound to the exit point of source(elem)
     ReducedPlugin :=ReducedPlugin ∪ elem
    elem := previous(elem)
  end while
  level:=UpMap(level)
  goto step2
end if
```

Table 5.2: Backward Slicing Algorithm

- A UCM construct and a UCM component: The resulting UCM slice obtained by applying the backward slicing algorithm is further reduced by keeping only UCM constructs bound to the underlined component.
- A UCM component: The resulting UCM slice is obtained by a simple projection of the UCM specification based on the component name.

The use of UCM component part of the slicing criteria may result in more than one non executable slice. Indeed, there is no guarantee that all UCM constructs that belong to one specific component are causally related.

5.2.7 Solving the Reachability Expression

The resulting UCM slice is considered to be correct, if and only if the set of computed conditions are satisfied. Given a reachability expression the question is: *Exist there any true/false assignments that will change the entire expression to true?*

- **The Boolean Satisfiability Problem (SAT).** Since UCM deals only with boolean variables, the reachability problem can be reduced to an instance of the Boolean Satisfiability Problem (SAT) [Coo71]. SAT is the first known NP-complete problem, as proved by *Stephen Cook* [Coo71] in 1971. There are many approaches for solving instances of SAT in practice. Just to name few: Davis-Putnam, GRASP, WALKSAT, GSAT, CHAFF and SATO. Finding a solution to the reachability expression is outside the scope of this thesis. For a detailed coverage of this problem refer to [GPFW97].
- **Conflicting conditions and non-determinism.** We may obtain unsatisfiable reachability expressions in the following situations:
    1. **Conflicting conditions:** unsatisfiable set of conditions in successive alternatives found in OR-Forks (For example: C1 and $\neg$C1), in selection policies of nested dynamic stubs, etc.
    2. **Non-determinism:** UCMs may contain some non-deterministic behavior due to overlapping conditions (For example: in an OR-Fork, conditions *Cond1:(C1=true)* and *Cond2: (C1=true and C2=true)* overlap when *C2=true*. This will result in a non-deterministic execution. Hence, the resulting initial condition does not guarantee the execution of the computed slice.

    Note: Parnas tables can be applied at specification time to determine, if a collection of conditions is deterministic and complete [PMI94].

5.2.8 UCM Data Flow

- **Variable Assignment.** So far, global boolean variables were assigned values only at initialization time. However, UCM responsibilities may affect the content of value identifiers. As a result, the reachability expression may not hold and the correctness of the computed slice is affected.

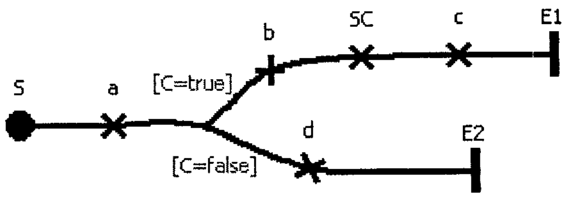


Figure 5.3: Responsibilities Updating Boolean Variables

- **Case 1**: Suppose that in the UCM of Figure 5.3, responsibility a:C$\longleftarrow$ ¬C. Consequently, the new definition of variable C should be considered in the reachability expression : C=*true*, C $\longleftarrow$ ¬C.
- **Case 2**: Suppose that in the UCM of Figure 5.3, responsibility b:C$\longleftarrow$ ¬C. The update happened after a path has been taken. The reachability expression should not be affected and should remain: C=*true*.

This mixture of predicates and assignment statements should be eliminated before applying a satisfiability algorithm[GPFW97]. In order to obtain a reachability expression containing only predicates, we substitute the affected variable of the assignment statement in the logical predicates(also called *unification*). For example: C=*true*, C$\longleftarrow$ ¬C $\Longrightarrow$ *true*=¬C. This problem is formalized and solved by the two following rules:

Rule 1 *If a variable has been assigned a new value before participating in a choice condition, then the variables of the choice are substituted with the new variable assignment.*

$$v \longleftarrow f(x_1,\ldots,x_n),\ g(y_1,\ldots,y_n,v) \Longrightarrow g(y_1,\ldots,y_n,f(x_1,\ldots,x_n))$$

where v is a boolean variable, f and g are logical expressions.

Rule 2 *If a variable has been assigned a new value after participating in a choice condition, the predicate condition is retained in the reachability expression and the assignment is ignored.*

$$g(y_1,\ldots,y_n,v), v \longleftarrow f(x_1,\ldots,x_n) \Longrightarrow g(y_1,\ldots,y_n,v)$$

where v is a boolean variable, f and g are logical expressions.

- **Limitations.** While the underlined rules are easy to apply and help reducing the reachability expression, they are not applicable in the following circumstances:

  1. **Loops:** When a UCM contains loops, the number of times a loop is visited is known only at run time. Such information, which depends on the variable's initial values and guard's evaluation, is needed in order to compute the slice and to solve the reachability expression.

For example, in the simple UCM of Figure 5.4(a), the number of times the loop is entered (zero or one time) is not available when the backward traversal is performed.

In a more complex situations, where instead of having only a responsibility like R2 in Figure 5.4(a), we have a dynamic stub, where the selection of plug-in maps depends on the values of the variables at run-time. Hence, non executable plug-in maps may be part of the resulting slice, whereas they should be left out.

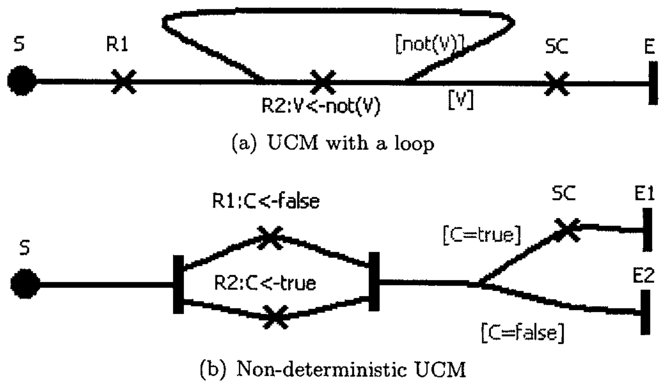


Figure 5.4: UCM Data Flow: Special Cases

2. **Non-determinism:** Figure 5.4(b) shows a UCM with two interleaving responsibilities R1 and R2. SC is reached only when R2 is executed after R1. One possible option is to investigate all possible alternatives (i.e. execution paths). Each alternative will be evaluated separately and considered in the resulting slice if it is a consistent one. Therefore, the resulting slice will be the union of all consistent executions. Another option is to keep the non-determinism. Users can then further analyze the resulting slice and make the appropriate decision.

5.2.9 UCM Forward Slicing

In what follows, we present our two phase UCM forward slicing algorithm. The first step consists on localizing *SC*. the 2nd step consists of a forward traversal of the specification starting from *SC*. A transition closure is applied to the transition relation λ staring from the tuple containing *SC*. We obtain as result a reduced transition relation containing only UCM transitions occurring after the execution of *SC*. In order to obtain a valid reduced UCM, we add a start point (referred to as *S*) to the obtained slice. Table 5.3 describes the high level schema of the UCM forward slicing algorithm.

5.3 Change Impact Analysis

Fixing specification errors may introduce new collateral errors. *Change Impact Analysis* technique is used to assess the impact of a change and to prevent the introduction of new errors.

```
Input:UCM Spec + (Slicing criterion SC, level, stub)
Output: Reduced UCM Spec

Step1:(* Searching SC *)
while (source(elem) ≠ SC) do /* elem of type UCMElement */
        elem := next(elem) /* next is defined such that target(elem)=source(next(elem)) */
end while
if (level=RootMap) then
    ReducedSlice := (S, e, SC) ∪ elem
    goto step2
else
    ReducedPlugin := (S, e, SC) ∪ elem
end if
Step2:(* UCM Forward Traversal *)
if (level=RootMap) then
  while (type(target(elem))≠ EndPoint) do
    ReducedSlice := ReducedSlice ∪ elem
    elem := next(elem)
  end while
   Compute transition closure else
  while (type(target(elem))≠ EndPoint) do
    ReducedPlugin:=ReducedSlice ∪ elem
    elem := next(elem)
  end while   level:=UpMap(level)
  goto step2
end if
```

Table 5.3: Forward Slicing Algorithm

Impact analysis techniques can be partitioned into two categories: traceability analysis and dependence analysis [Arn96]. Dependence-based impact analysis found in [CR94, GPFW97, PC90, RT01] attempts to assess the resulting changes on semantic dependencies among program entities. This is done by identifying the syntactic dependencies that may signal the presence of such semantic dependencies [AB93]. The techniques used to identify these syntactic dependencies include static [Wei84] and/or dynamic [KL88] slicing techniques. Other techniques using transitive closure on call graphs [Arn96] attempt to approximate slicing-based techniques, while avoiding the cost associated with dependency analysis. Approximate dependence-based impact analysis techniques include expert judgment and code inspection. These approaches may often be incorrect [LS98], and performing impact analysis by inspecting source code can be expensive [Pfl01], due to a lack of automation.

In our validation method, we focus on combining a UCM forward slicing algorithm with the dependency analysis techniques introduced in this section to address some of the shortcoming of the existing approaches [Arn96, GPFW97, Wei84]. The dependency analysis algorithm uses as input a UCM specification and the necessary slicing criterion (based on the change request) that will provide the set of impacted UCM elements. In what follows we provide a detailed discussion on UCM based dependency analysis at the scenario level.

5.3.1 UCM Scenario Dependencies

Scenarios in UCM inherently contain dependency information as part of their modeling. Scenario dependencies can be applied in assessing the ripple effects of a change at the scenario level of understanding. At the UCM level of abstraction, we distinguish three types of dependencies: functional, containment and temporal.

- **Functional dependency.** UCMs integrate many individual scenarios. We can define system level scenarios as being end to end scenarios, where each scenario starts at a start point and ends at an end point. Scenario definitions are used to describe particular scenarios, representing them as partial orders of UCM elements (i.e. sequence and concurrency are preserved, but alternatives are resolved). System level scenarios make use of a path data model composed of global variables used on guarding conditions. A scenario definition contains an identifier, a name, initial values for the global variables, a list of start points, and (optionally) post-conditions expressed using the global variables.

Scenario Group	Number	Scenario Name	variables			
			Busy	OnOCSList	subCND	subOCS
Basic call	1	BCbusy	T	-	F	F
	2	BCsuccess	F	-	F	F
OCS feature	3	OCSbusy	T	F	F	T
	4	OCSdenied	F	T	F	T
	5	OCSsuccess	F	F	F	T
CND feature	6	CNDdisplay	F	-	T	F
OCS_CND	7	OCS_CNDdisplay	F	F	T	T

Figure 5.5: System Scenarios Definitions

From the case study of the telephone system introduced in Section 2.1.4, one can identify seven system level scenario definitions that are summarized in Figure 5.5. All scenarios start at the start point req. Variables *Busy* and *InOCSList* are used to guard the two OR-forks found in the plug-in maps, whereas *subOCS* and *subCND* are used to define the selection policies found in the dynamic stubs. No postconditions are necessary here. These scenarios cover all the paths found in this UCM model and they are organized in functional groups. Functional dependencies capture the coexistence of two or more scenarios inside a same conceptual (or logical) cluster. For instance, we have grouped scenarios according the features they are describing.

- **Containment dependency.** A containment dependency exists between a scenario S2 and a scenario S1, if S2 is used in the description of S1. Stub plug-in maps are contained in system level scenarios since they describe disjoint pieces of the system scenarios. For instance, CND plug-in is part of the terminating plug-in which is part of all system level scenarios except the *OCSdenied*. Default plug-in is part of both "Originating plug-in" and "terminating plug-in". Figure 5.6 illustrates the containment dependency graph.

- **Temporal dependency.** Temporal dependency capture different types of temporal relationships that may exist between scenarios (e.g., one scenario excludes, waits for, aborts, rendezvous or joins another, concurrent, mutually exclusive, etc.). For the sake of generality

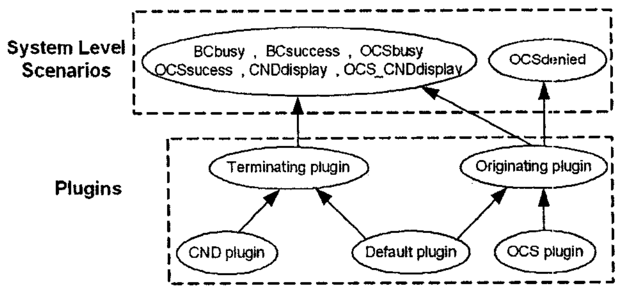
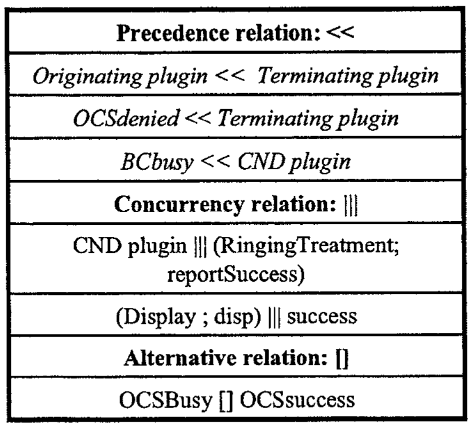


Figure 5.6: Scenario Containment Dependency

temporal dependencies may be defined between system level scenarios, plug-in maps or even sequential pieces of behavior. We denote the precedence relation by "$\ll$", the concurrency relation by "|||", the alternative relation by "[]", abort relation by "[>" etc. Figure 5.7 illustrates some examples of temporal dependencies between scenarios of the simple telephone system.



Precedence relation: <<
Originating plugin << Terminating plugin
OCSdenied << Terminating plugin
BCbusy << CND plugin
Concurrency relation: |||
CND plugin ||| (RingingTreatment; reportSuccess)
(Display ; disp) ||| success
Alternative relation: []
OCSBusy [] OCSsuccess

Figure 5.7: Temporal Dependencies

5.3.2 Ripple Effect Analysis

Impact analysis techniques based on source code analysis have the clear advantage of being very accurate in the analysis as they identify impacts in the final product; however, they have the disadvantage of being very time consuming, limited in scope, and they require implementation of the change before the impact can be determined [BLO03].

Change impact analysis, also often referred to as ripple effect analysis, is generally performed after the change has been implemented [Arn96]. However, during change impact analysis, it is useful to see the potential effect that performing a change might have on the rest of the system. Ripple effect analysis is an iterative process which continues until no further ripples can be determined.

We apply the UCM slicing algorithm to the UCM specifications to determine the ripple effect of a change.

For component ripple effect determination the output is the set of components that are related to the change component through its scenario paths. The execution of the slicing algorithm adds to the impact set any new components that are encountered along the execution path. This impact set contains all the components that relate to the change component through any of the scenario paths that it is contained within.

5.3.3 Related Work: Model Based Change Impact Analysis

Lehman provides an in-depth analysis of different aspects of software evolution in [LR01]. He addresses the different types of systems and how they evolve; the evolution of the system in its context; the evolution of the development process. Requirements evolution is highly focused on tracing changed requirements to design, but there is little mention of how to assess the impact of changes at the requirement or design level. Requirements change analysis is discussed in [SS96] with a focus on assessing the information and techniques useful in assessing the risk of a changed requirement. Both sensitivity analysis and impact analysis are needed in a pro-active approach to change analysis. Settimi et al. present in [SCHK+04] their work on software evolution, with a similar aim than ours - to provide a higher level of understanding of the change impact. However, they focus on Information Retrieval (IR) methods to facilitate traceability analysis to UML models. Similarly in [vK01] a fine-grained trace model for requirements impact analysis in embedded systems is presented.

Bai et al. [BTF+02] propose a scenario-based functional regression testing. In their approach, they have also integrated scenario based ripple effect analysis, traceability information, and slicing to determine affected components. Their focus however is on identifying components that have to be retested, limiting the analysis to a subset of the slicing criterion supported by our approach. Furthermore their approach requires the availability of source and being able to create traceability links between scenarios and source code.

Ecklund et al. [EDF96] propose the notion of change cases, an adapted version of use cases, to be developed and maintained at the time of design to identify and incorporate expected future changes into the design to enhance the long-term robustness of the design. This idea provides an idealistic view, since it assumes that there are no time constraints on the development, and that it is possible to provide a conclusive prediction of future requirement changes. Furthermore it also requires that the change cases are maintained during the software and design evolution.

Briand et al. [BLO03] propose a change impact analysis method that is based on UML models and can be applied before implementing the changes. They have defined impact analysis rules to determine the directly and indirectly affected model components that depend on the type of change for which the impact analysis has to be performed. They also defined one rule for each change type. This approach focuses on defining rules using OCL that can be used on static UML models to formally determine the impact of a change. As well, very detailed UML models are used in this approach, requiring that detailed design descriptions are completed. Furthermore the approach focuses on the functional requirements rather on design changes.

5.4 Applying the Validation Approach

In this section, we apply the proposed validation approach on the simple telephone system presented in Section 2.1.4.

5.4.1 Simple Telephone System Traces

Suppose that we want to validate the behavior of *CND* feature in isolation and later validate its behavior in presence of *OCS* feature. Figures 5.8 and 5.9 show two execution traces of the telephone system with the callee party subscribed to *CND* only. Each scenario provides the initial values of the specification variables.

```
This scenario is generated with the following
initial values:
subCND:True
subOCS:False
busy:False
================================
Start Executing: Telephone System:Req
Start Point:Req in Component:UserOrig
Stub_Construct: SOrig
Plugin: Orig_plugin
Start Point:Start in Component:AgentOrig
Stub_Construct: Sscreen
Plugin: DEF_Plugin
Start Point:Start in Component:AgentTerm
End point: continue in Component:AgentTerm
Responsibility:  snd_req  in component: AgentOrig
End point:  success  in Component:AgentOrig
Stub_Construct: Sterm
Plugin: term_Plugin
Start Point:Start in Component:AgentTerm
OR-Fork:  term_OF1
AND-Fork: term_AF1
Stub_Construct: Sdisplay
Plugin: CND_Plugin
Responsibility: ringingTreatment in component:
AgentTerm
End point: reportSuccess in Component:AgentTerm
Responsibility: fwd_sig in component: AgentOrig
End Point:  ringing part of root map reached in
Component:UserOrig
Start Point:Start in Component:AgentTerm
AND-Fork:  CND_AF1
Responsibility: display in component: AgentTerm
End point: success in Component:AgentTerm
Responsibility: ringTreatment in component:
AgentTerm
End point: success in Component:AgentTerm
End point: disp in Component:AgentTerm
End point: display in Component:AgentTerm
End Point: display part of root map reached in
Component:UserTerm
End Point: ring part of root map reached in
Component:UserTerm
```

Figure 5.8: CND Trace (with busy = False)

```
This scenario is generated with the following
initial values:
subCND:True
subOCS:False
busy:True
================================
Start Executing: Telephone System:Req
Start Point:Req in Component:UserOrig
Stub_Construct: SOrig
Plugin: Orig_plugin
Start Point:Start in Component:AgentOrig
Stub_Construct: Sscreen
Plugin: DEF_Plugin
Start Point:Start in Component:AgentTerm
End point: continue in Component:AgentTerm
Responsibility:  snd_req  in component:
AgentOrig
End point:  success  in Component:AgentOrig
Stub_Construct: Sterm
Plugin: term_Plugin
Start Point:Start in Component:AgentTerm
OR-Fork:  term_OF1
Responsibility:  busyTreatment  in component:
AgentTerm
End point: fail in Component:AgentTerm
Responsibility: fwd_sig in component:
AgentOrig
End Point: busy part of root map reached in
Component:UserOrig
```

Figure 5.9: CND Trace (with busy = True)

Figures 5.10 and 5.11 show two execution traces of the telephone system with the callee party subscribed to *CND* and the caller subscribed to *OCS*.

```
This scenario is generated with the following initial values:
subCND:True
subOCS:True
InOCSList:False
busy:False
------------------------------------------------------------
Start Executing: Telephone System:Req
Start Point:Req in Component:UserOrig
Stub_Construct: SOrig
Plugin: Orig_plugin
Start Point:Start in Component:AgentOrig
Stub_Construct: Sscreen
Plugin: OCS_plugin
Start Point:Start in Component:AgentOrig
Responsibility: checkOCS in component: AgentOrig
OR-Fork: OCS_OF1
End point: success in Component:AgentOrig
Responsibility:  snd_req  in component: AgentOrig
End point:  success  in Component:AgentOrig
Stub_Construct: Sterm
Plugin: term_Plugin
Start Point:Start in Component:AgentTerm
OR-Fork:  term_OF1
AND-Fork: term_AF1
Responsibility: ringingTreatment in component: AgentTerm
Stub_Construct: Sdisplay
Plugin: CND_Plugin
End point: reportSuccess in Component:AgentTerm
Start Point:Start in Component:AgentTerm
AND-Fork:  CND_AF1
End point: success in Component:AgentTerm
Responsibility: display in component: AgentTerm
Responsibility: ringTreatment in component: AgentTerm
End point: success in Component:AgentTerm
End point: disp in Component:AgentTerm
End point: display in Component:AgentTerm
End Point: ring part of root map reached in Component:UserTerm
End Point: display part of root map reached in
Component:UserTerm
Responsibility: fwd_sig in component: AgentOrig
End Point:  ringing part of root map reached in
Component:UserOrig
```

Figure 5.10: CND-OCS Trace (with InOCSList = False)

```
This scenario is generated with the
following initial values:
subCND:True
subOCS:True
InOCSList:True
busy:False
========================
Start Executing: Telephone System: Req
Start Point:Req in Component:UserOrig
Stub_Construct: SOrig
Plugin: Orig_plugin
Start Point:Start in Component: AgentOrig
Stub_Construct: Sscreen
Plugin: OCS_plugin
Start Point:Start in Component: AgentOrig
Responsibility: checkOCS in component:
AgentOrig
OR-Fork: OCS_OF1
Responsibility: deny in component: AgentOrig
End point: fail in Component: AgentOrig
End point: fail in Component: AgentOrig
End Point: notify part of root map reached
in Component:UserOrig
```

Figure 5.11: CND-OCS Trace (with InOCSList = True)

5.4.2 CND Feature upgrade: A Closer Look

Suppose that we want to perform an upgrade to the *CND* feature. The upgrade will involve the display not only of the caller's name but also of his/her service provider. This maintenance task cannot take place until the maintainer understands how the particular feature works and how it interacts with other system features. Knowing all the details of the requirement specification is almost never necessary; an experienced maintainer will try to extract only just enough information to perform the task at hand. The goal is to extract the scenarios leading to the display function. Hence, the slicing criterion is the responsibility *display*. Figure 5.12 describes the resulting UCM obtained from the original UCM of Figure 2.9 with respect to the slicing criteria *display*.

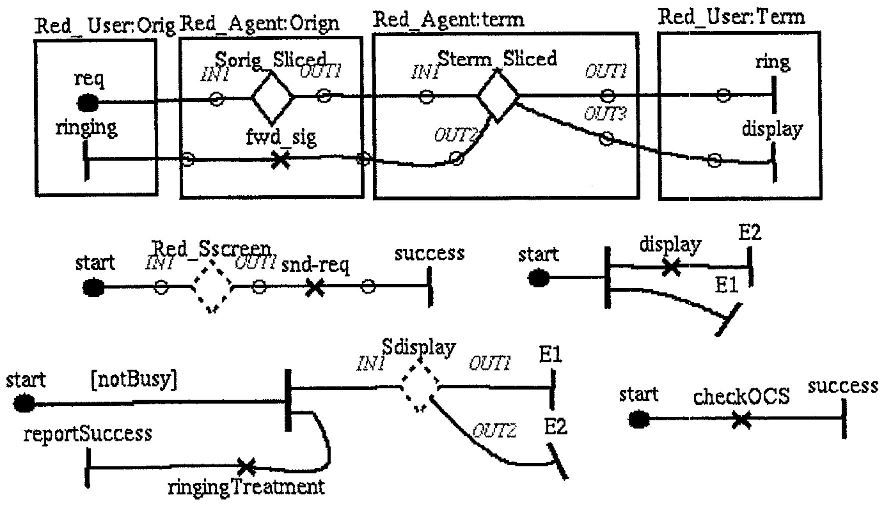


Figure 5.12: Simple Telephony System Slice with Respect to SC:*display*

Figure 5.13 shows its corresponding *Reachability Expression.* The first part of the reachability expression ((1) in Fig 5.13) illustrates the fact that the default plug-in is selected *(subOCS = false)* and the second part of the expression ((2) in Fig 5.13) expresses the fact that the OCS plug-in was selected.

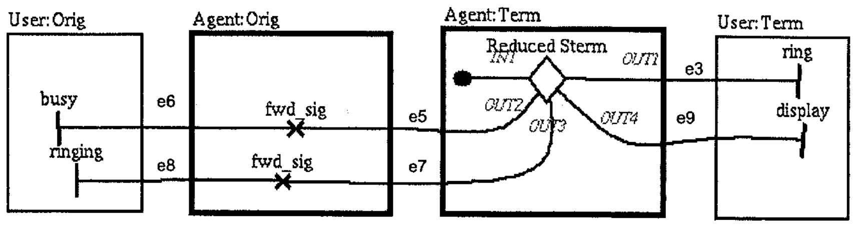
((subCND = true) AND (Busy =false) AND (subOCS = false)) (1) OR ((subCND=true) AND (Busy=false) AND (subOCS=true) AND (InOCSList=false)) (2)

Figure 5.13: Reachability Expression for Responsibility *display*

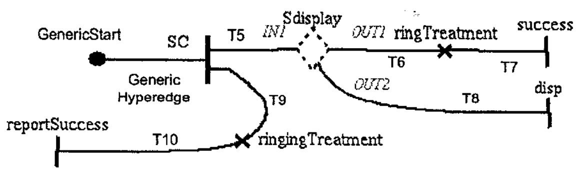
In our example the reachability expression provides the initial values of global variables leading to the slicing criterion and no further computation is needed.

5.4.3 Simple Telephone System: Change Impact Analysis

Suppose that the system's maintainer wants to assess the possible impact of changing the semantics of the Terminating plug-in, for instance changing the *AND-Fork* by an *OR-Fork*. The UCM Slicer computes the UCM forward slice with the *AND-Fork* of the terminating plug-in as slicing criterion. The resulting forward slice is described in Figure 5.14. All the four components are impacted by this change. It can be observed that the resulting forward slice no longer includes stub *Sorig* within *Agent:Orig*, also the corresponding start and end points in the *User:Orig* component are excluded from the slice. As with traditional program slicing approaches [KL88, Wei84], the slice size is directly affected by several factors. As shown in [BH03] the slicing criteria, its position within a scenario/component and the overall cohesiveness of the system play an important role for the slice reduction. Furthermore, compared to the more traditional program slicing techniques, where the slicing criterion is restricted to a single variable, our UCM slicing approach supports multiple types of slicing criteria at different levels of granularity, that are also influencing the slice size.



(a) Reduced Root Map



(b) Reduced terminating plug-in

Figure 5.14: Telephony System Slice for SC=AND-Fork

Substituting the AND-Fork by an OR-Fork will alter the stored temporal dependencies. For example, scenarios "CND plug-in" and "RingingTreatment; reportSuccess" behave now as alternatives not concurrently as initially described in figure 5.7. Hence, both parts composing the temporal dependency (i.e., both scenarios) may be affected and then should be investigated with respect to functional and containment dependencies.

On the one hand, based on the identified containment graph (see Figure 5.6), the *CND* plug-in is enclosed in all system level scenarios (except *OCSdenied*) and in the terminating plug-in as well. All these identified scenarios may be affected by the planned change and should be communicated to the user. On the other hand, the introduction of the new *OR-Fork* may also alter the table of functional dependencies (see Figure 5.5). In our example, the maintenance task has also an effect of increasing the number of functional dependencies within the system since by applying our scenarios definitions alternatives are resolved whereas concurrencies are preserved. Therefore new system level scenarios have to be defined. Once the user commits the changes these new scenarios are integrated in the Scenario Dependency Manager.

5.5 Chapter Summary

In this chapter, we have presented our *Early Stages Validation Approach* which combines validation techniques (i.e., step by step simulation, trace generation) with slicing techniques to validate early requirement specifications described using the Use Case Maps notation.

Section 5.1 described our high level validation approach. In Section 5.2, we extend traditional program slicing to functional requirement specifications. This new application of slicing, called *UCM Requirement Slicing* helps reduce the complexity of the requirement specifications and facilitates requirement comprehension and maintenance. Our approach is *two tiered*: First, we allow an analyst to reduce a UCM specification according to a slicing criterion. Second, a reachability expression is attached to the slice, which insight on the feasibility of the selected scenarios. However, the choice of the appropriate slicing criteria remains the big challenge that a designer/maintainer have to face. Indeed, not choosing the *right* slicing criterion may lead to chopping parts of the specification that contain design flaws, leaving them uncovered.

Furthermore, we have illustrated the potential use of UCM slicing in assessing the impact of a change (i.e., fixing errors, upgrade a feature, add a new feature, etc.). Moreover, we see potential application domains of UCM slicing in feature extraction and reuse of requirements. In fact, while reuse of code is important, more significant improvements in productivity and quality can be expected from reuse of software designs and requirement patterns. By slicing a UCM requirement specification, a system designer can extract reusable parts from it, and reuse them into new system designs for which they are appropriate.

Finally the proposed approach is not limited to Use Case Maps specifications. The approach is general enough to be applied to all languages with guarded transitions such as UML activity diagrams.

Chapter 6

Timed Scenario Languages

The ability to perform quantitative analysis at the requirements level supports the detection of design errors during the early stages of a software development life cycle. Thus, reducing the cost of later redesign activities in case some of the required time constraints, for instance, are not met. In order to achieve this goal, non-functional aspects and in particular time-related aspects have to be incorporated at the software requirement phase. This is essential in order to correctly model time dependent applications at early stages in system development. Typical classes of such applications are communication protocols and real-time distributed systems.

In this chapter, we survey thirteen timed scenario notations and we propose a collection of eleven criteria to categorize and compare various timed notations.

6.1 Evaluation Criteria

In Chapter 3, we have presented some classification approaches of untimed scenario notations [AE03, LDD06, RAC+98, Coc97, AEG+98]. In this section, we focus on the timing aspects of scenario notations. We propose a collection of eleven criteria that will help categorize and compare many timed scenario notations.

6.1.1 Timed Action/Event Enabling

Intuitively, an action/event is enabled (i.e. offered) when the execution of its predecessor is completed. However, a time constraint can be specified to define when an action/event is offered relative to the execution of its predecessor action/event. Three types of enabling can be defined [BG06]:

- Simple enabling: The instant an action becomes enabled may be associated with a time constraint. For example, an action b can be taken at any time 5 time units after action a. However, no upper bound can be imposed on enabling.

- Initiation and termination of enabling: both lower and upper time bounds can be imposed on enabling. Thus, an action is offered within a specific time interval. For example, an action a can be enabled immediately and will be offered for 5 time units.

- Punctual enabling: This type is a restriction of the *Simple enabling.* An action/event is enabled with respect to its associated time constraint then the enabling retracts if the action is not taken. For example, an action *b* is offered 5 time units after *a* and should be taken instantaneously, otherwise the enabling is retracted. Punctual enabling are used as abstractions of real-world systems. An important class of applications that use punctual enabling are those employing periodic behavior.

Another notational alternative to express enabling classes is:
Delays: An action/event can be explicitly delayed by using a delay operator, such as δt or *WAIT*t. Using such operator, lower bounds on enabling can be defined. In their work on timed process algebra, Leonard and Leduc's motivation for including an explicit delay operator is to enable delays to be imposed that do not resolve the choice on expiry of the delay [LL93b]. For example, the following behavior (δ5 B) [] (δ7 B') imposes 5 time unit (respectively 7) delays on B (resp. B'), but it is important to note that the choice between B and B' cannot be resolved solely by the expiry of either of the delays.

6.1.2 Instantaneous (atomic) vs. Durational Actions

Actions can take a given amount of time, called *duration*, to be performed [GRS95, CFP01]. Hence, the time passes due to the execution of these actions. Approaches that use durational actions may support true concurrency. Alternatively, actions can be instantaneous (atomic) and the passage of time is explicitly modeled by a special *tick* action [RR88, NS94].

In an interleaving semantics, concurrency is reduced to *non-determinism* where the behavior of a system that performs two actions *a* and *b* concurrently is considered the same as the behavior of a system that either does an *a* followed by *b*, or a *b* followed by *a*. However, in the context of durational actions, Hoare [Hoa85] suggests that time-consuming actions should be represented by a pair of events, the first denoting its start and the second denoting its finish.

6.1.3 Relative vs. Absolute Time

The time of occurrence of an action/event of a system execution can be related to the value of the global clock, in this case, the time features as *absolute.* Alternatively, they can be relative to the execution of a causally preceding action, in this case, the time features as *relative.* In this case, the preceding action/event enables (directly or indirectly, i.e. via some intermediate events) the subsequent action/event.

6.1.4 System Clocks: Local vs. Global, Physical vs. Logical

The elapsing of time in a system can be modeled by a unique centralized global clock that increases uniformly. However, in a distributed system, many local clocks are used to track time in different locations. This raises the problem of clock synchronization [Mes90]. To address this issue, Lamport [Lam78] has introduced the concept of logical clocks, so partial ordering of events can be obtained without recourse to any physical "real" time.

6.1.5 Urgency

Urgency is a well-accepted and extensively documented time requirement [NS92, HR95, Sin04, BST98, BS00]. Urgency offers an abstraction that can be used to influence the behavior of a system as time progresses. It allows for expressing assumptions on the environment and on the underlying execution system, such as action durations, communication delays, or time constraints on external inputs. We distinguish two main approaches:

1. *Action Urgency.* This type of urgency is studied extensively in the process algebra theory. Three main approaches have been identified [BG06]:

   - *Urgent Actions.* In this approach, all observable and unobservable actions (i.e. internal to the system) are interpreted as urgent. Hence, it is possible that observable actions will become urgent, but will be prevented from executing by an environment that is not offering the action. This leads to a *timelock* situation in which time is not able to pass. Urgent actions are largely rejected in timed process calculi.

   - *Explicit Urgency Operator.* In this approach, a specific operator, such as *urge* [BL92], is used to make an action urgent. *urge* is associated to an action and is placed in the beginning of a process behavior. This approach constraints the environment more selectively but timelocks can still occur [BG06].

   - *Urgent Internal Actions.* This approach restricts urgency to internal actions (i.e. unobservable to the environment) and all observable actions are interpreted as non-urgent. Internal actions can always execute urgently without the possibility of *timelock* since they are not controlled by the environment. This approach is now the most common approach in timed process algebra [ISO97]. Variants of this class of urgency are *Maximal progress*, *asap*(as soon as possible) and *Minimal Delay.*

2. *Transition Urgency.* A transition can be regarded as urgent, if it will be taken or disabled before time progresses. There exist three main types of transition urgencies:

   - *Eager transitions.* Eager transitions are urgent as soon as they are enabled, i.e. they never wait. They have to be executed as soon as possible and time should not progress as long as an eager transition is enabled.

   - *Lazy transitions.* Lazy transitions do not prevent time progress in any system state, i.e. they can wait. Whenever a lazy transition is enabled, it can be taken, or likewise time can progress and possibly disable it.

   - *Delayable transitions.* Delayable transitions are a combination of both eager and lazy transitions. They can wait, but they become urgent when time progress would disable them.

The distinction between these three types of transitions is depicted in Figure 6.1 [Sin04].

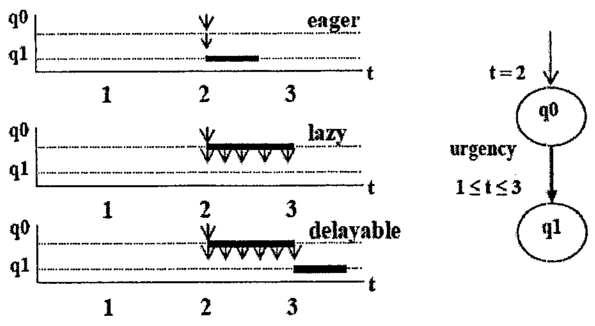


Figure 6.1: Transition Urgencies [Sin04]

6.1.6 Time Domain

The expression *discrete* or *continuous time* often refers to the empirical description of a so-called *physical time.* There are three types of physical time:

- *Discrete Time.* Empirical time is composed of indivisible instants, such that the passage from one instant to another implies an irreducible jump. In this sense, discrete time is a model isomorphic (i.e. structurally identical) to a discrete series of natural numbers. This type of time model is appropriate for *synchronous systems*, where all the components are synchronized by a single common clock. This model has been successfully used for reasoning about the correctness of synchronous hardware design especially synchronous digital circuits, where signal changes are considered to change exactly when a clock signal arrives. One of the advantages of this model is that it can be transformed easily into an ordinary formal language. Each timed trace can be expanded into a trace where the times increase by exactly one at each step, by inserting a special *silent* event as many times as necessary between events in the original trace. Once this transformation has been performed, the time of each event is the same as its position [EMCGP99].

- *Continuous Time.* The jump between two instants is a smooth and uninterrupted process. In this sense, the model of continuous time is isomorphic to a continuous series of non negative real numbers. Continuous time model is appropriate for *asynchronous systems*, because the separation of events can be arbitrarily small. This ability is desirable for representing causally independent events in an asynchronous system. Moreover, no assumptions are needed about the speed of the environment when this model of time is assumed.

- *Dense (but countable) Time.* A model of dense time is isomorphic to a dense series of non negative rational numbers, meaning that there is always a rational number between any two rational numbers.

6.1.7 Time Representation/Measurement

Time can be represented by three classes: *point-based*, *interval-based* or both of them. In point-based models, the elementary units are points in a time space. Each event in the model has its associated time point (a single concrete time value). The time points arranged according to some relations such as *precede* or *after*. The ability to reference the time instant at which an action/event occurs is important in certain classes of real time specifications. In interval-based models, two different approaches can be considered. In the first, intervals are assumed to consist of points, and hence, the corresponding systems may be considered as models of point-based time theories [KM94a]. The second approach takes intervals (i.e. ranges of time values within given bounds) as primitive objects, without any reference to the definitions of internal-point structures. Interval-based models are mainly based on the relations defined by Allen [All81]: before, meets, overlaps, finishes, during, starts, equals.

Time observations are described by measurements. Measurements are used to observe the delay between the enabling and occurrence of an event/action (for relative timing) and to measure the absolute time of the occurrence of an event/action (for absolute timing).

6.1.8 Timed Constructs/Constraints

A timed scenario requirement language is expected to offer a set of constructs that help:

- Express time dependent system behavior, such as execution times of tasks and actions. These are often modeled by means of timers or explicit access to a system clock.

- Express time constraints on the internal system execution such as end-to-end delays of the system.

- Express time related assumptions on the external environment of the system, mainly response times and inter occurrence times of stimulus.

6.1.9 Formal vs. Semi-formal Semantics

The nature of semantics (formal or semi-formal semantics) offered by an approach, as well as their expressiveness power represents an important and useful classification criterion.

6.1.10 Time Analysis and Verification

The quantitative analysis of requirement models allows the early detection of potential behavioral (time dependent) and performance problems. This criterion aims to identify what kind of timing analysis (such as validating timing assignment, verifying timing consistency, etc.) scenario notations offer.

Different approaches have proposed algorithms and methods to analyze and ensure that timing requirements are met. These approaches can be classified as follows:

- Model-Based Scheduling Analysis: The first contribution to real-time scheduling theory was made by Liu and Layland [LL73], who developed optimal static and dynamic priority scheduling algorithms for hard real-time sets of independent tasks. Since then, much work on schedulability analysis has been done which includes various extensions of these results [HKL94, TBW94, JP86]. Schedulability analysis techniques are based, amongst others, on worst case execution times (WCET) [PB00] and stochastic task execution times [GL99].

  Note: End-to-end system behavior description is necessary to conduct schedulability analysis.

- Formal Verification Approaches: These approaches are based on a translation of the timed requirement model into a formal description technique supporting time, such as timed automata [AD94]. The resulting models are checked against timing requirements using formal verification techniques, such as model checking [EMCGP99].

6.1.11 Specification Executability and tool support

Using scenario approaches to describe timing requirements may lead to:

- Executable specifications with appropriate operational semantics that can be simulated and tested.

- Off-line specifications that are not testable but offering a rich expressive power of time constraints.

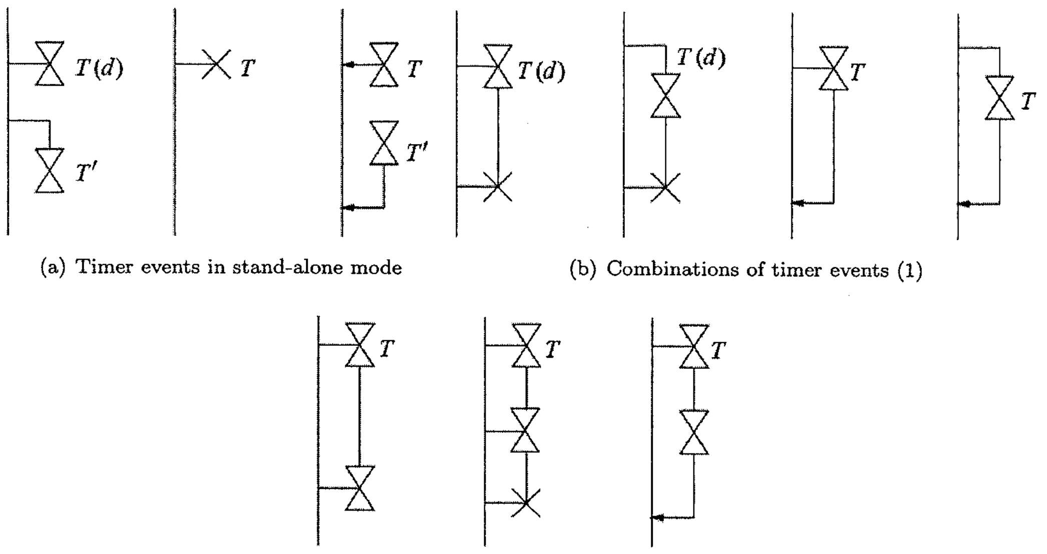
6.2 Survey of Selected Timed Scenarios Languages

In Section 3.3, we have presented a literature review of untimed scenarios classification approaches [LDD06, AE03, RAC+98, Coc97, AEG+98]. In this section, we focus our survey on selected timed scenario languages.

6.2.1 Timed (variants of) MSCs

Basically, timing constraints in (variants of) MSCs notations are expressed using *timers* [IT96, AHP96], *delay intervals* [AHP96, MS93] and *timing markers* [GBJ96, LL99a].

- *Timers.* Timer support in the early standard version of MSC language (MSC-96 [IT96]) is very basic. A timer can be set to an optional duration, reset to zero, and observed for timeout. Figure 6.2 illustrates the stand-alone occurrences of the timer events in MSC-96 standard as well as combined timer events. A timer set event (labeled by the timer name and with an identifier for the duration) is denoted by an hourglass symbol attached to the instance axis by means of a horizontal or bent line. A timer reset event is denoted by a cross which is attached to the instance axis by means of a horizontal or bent line. A timeout is represented by an hourglass symbol which is attached to the instance axis by means of an horizontal or bent arrow from the hourglass symbol to the instance axis.



(a) Timer events in stand-alone mode

(b) Combinations of timer events (1)

(c) Combinations of timer events (2)

Figure 6.2: Timer Handling in MSC-96 [IT96]

The current standard (MSC-2004 [IT04]) describes some syntactic and semantic changes and refinements. The *set* event has been renamed to *start timer* and *reset* has been renamed to *stop timer*. Timer duration is specified with an interval having an optional lower bound and an optional upper bound allowing the timer to expire within the specified interval. The upper bound for a timeout can be defined to be infinity which is represented by the keyword *inf*. A timer can be used to express a maximal delay between two or more consecutive events in one process. A timer cannot be shared among concurrent processes in a basic MSC (*bMSC*).

- *Delay Intervals.* Delay intervals are used to express three types of timing constraints: (1) *event-associated intervals* [MS93] which are denoted as an interval that is as associated with an event (i.e. minimal and maximal delays within which the event should occur with respect to any previous event, whenever it occurs in an execution trace); (2) *message delivery delays* [AHP96, MS93] indicate the minimal and maximal delays allowed from the moment a message is sent until it is received (expressed as a time interval over a message arrow); and (3)*Processor's speed constraints* [AHP96, MS93] which are expressed as time intervals between two consecutive events along an instance line. In MSC-2004 standard [IT04], the delay between any pair of events can be constrained by defining a minimal or maximal bound for the delay between the two events. An interval must define at least one of the two bounds. An absolute time interval can be of the form [@1,@3) or @[1,3). Figure 6.3 shows an example of time constraints and measurement. A relative time measurement is used to observe the message duration of the *resolve_request* call (the time variable *rel1*). The measurement on the duration of the call is subsequently used to restrict the message duration for *resolve_reply*. The relative time constraint *(0,0.7*rel1]* allows the message to take at most 70 percent of the time it took to issue the call *resolve_request* from TC to SUT. In addition, the measurement on the duration of the call is used to constrain the execution of the instance TC: the relative time constraint *(rel1,3*rel1]* requires that after the output of the *requestNamedAccess* call, it takes at least *rel1* and at most *3*3rel1* to get the reply.

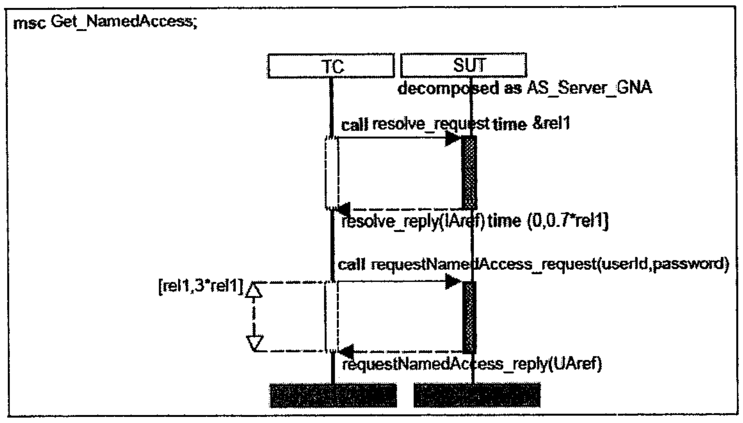

msc Get_NamedAccess;
TC
SUT
decomposed as AS_Server_GNA
call resolve_request time &rel1
resolve_reply(IAref) time (0,0.7*rel1]
call requestNamedAccess_request(userId,password)
[rel1,3*rel1]
requestNamedAccess_reply(UAref)

Figure 6.3: Time Constraints and Measurements [IT04]

- *Timing marks* is a boolean expression on event labels [GBJ96]. For instance the time marker a$\leq$ e_i-$e_j$$\leq$b, where e_i and e_j are event labels and a and b are real numbers, expresses that event e_j must occur within [a,b] time frame after event e_j. For basic MSCs, timing marks can be used to describe any timing constraints expressed by timers or delay intervals [LL99a].

In [MS93], the author have used the notion of consecutive events to generalize the message delivery and processor speed delay intervals. He extended the syntax of MSC with precedence edges that connect unrelated events and thus allow the user to provide delay intervals for them. These edges may result in a cumbersome and cluttered graph.

Li and Lilius [LL99a] define the behaviour of an MSC specification as the timed event sequences which are the concatenation of the timed event sequences representing the behaviour of the bMSCs which make up the High-Level MSC specification. Timing constraints are interpreted by *local semantics*: select one path at a time and analyze its timing requirements, independently of other paths that may branch out of the selected one. The authors [LL99a] provide an algorithm to decide about timing consistency.

Alur et al. [AHP96] interpret timed MSC as partial orders with timing functions that map each pair of events in the partial order to a time interval. In their timed MSC, time constraints can only be imposed on pair of events. They do not consider absolute time constraints at which events occur, and only bMSCs with sending and reception events are addressed. The authors provide also an algorithm for analyzing basic MSCs. Furthermore, the authors propose an MSC analyzer tool that offers timed analysis based on a semantics that accounts for the queuing strategies in a bMSC and hMSC. Similarly, Ben-Abdallah and Leue [BAL97] use timing delay intervals and timer events to express timing constraints. A MSC is interpreted as traces that are consistent with the partial order of events. They define a timing assignment that assigns a time stamp to each event in a trace. They also do not consider absolute time constraints. The authors augmented the timing analysis for bMSCs presented in [MS93] and [AHP96] to handle the possibility that a timer is set in a bMSC but not reset nor timeout follows the timer setting in the bMSC. This timing analysis is extended further with branchings and iterations. To address timing consistency, the authors proposed an approach that consists on translating bMSCs into a directed labeled graph, that they call, *temporal constraint graph*. Then this graph is checked to ensure that it didn't contain any cycles with negative cost.

[GDO98] define semantics of Timed MSCs in terms of Constraint Diagrams [Die96], a graphical notation for real-time properties stated in the Duration Calculus [CHR91].

MSC-2004 [IT04] standard assumes the following time concepts:

- Time progress (i.e. clocking) is equal for all instances in a MSC. Also, all the clock values are equal, i.e. a global clock is assumed.
- All events are instantaneous (i.e. atomic) and do not consume time.
- The time domain can be dense or discrete. It must be a total order with a least element, or origin, of time zero. It must be closed under an addition operation, used to compute time offsets.
- Time constraints can be used to specify the delay between any two events (relative delay), or to specify the time of occurence of an event (absolute delay). When specifying a relative delay, the time constraint can be an interval with minimal and maximal bounds or a concrete time value. Furthermore, Time constraints can be specified by the use of arbitrary expressions of type *Time*, i.e. referencing parameters, wildcards and dynamic variables.

In the MSC-2004[IT04] standard, the semantics of a timed MSC is represented by event traces with special time events between normal events. Hence, If there is no time event between two normal events, it means they occur simultaneously. Maigat et al. [LMH00] associate each pair of communication events in MSC with a duration. They propose partial order and (max,+) automaton based semantics and analysus for timed MSC considering HMSC and compositions.

Zheng et al. [ZKH02] provide formal semantics to timed MSCs in terms of *timed labelled partially ordered set* (lposet). First, they define the semantics of events as timed lposets. Then the semantics of bMSCs, MSCs with structures and hMSCs, are obtained using the operations defined on timed lposets. However, their semantics do not cover some MSC standard concepts such as general ordering, instance decomposition, gate and condition. In a related work [ZK02], the authors extend MSCs with a construct, called *instance delay*, to specify repeated MSC scenarios (i.e. specifies how long the scenario takes and the interval between the repetitions). The semantics of this construct is expressed in terms of labelled partially ordered set (*lposets*).

Kim et al. [KC06] proposed *timed high-level message sequence charts (THMSC)* which includes an unambiguous subset of time constraints and timed edges as a new complementary notation. Timed edges are directed time constraints between two consecutive MSCs.

They claim that *THMSC* is effective in accurately specifying popular requirement patterns such as watchdog timers and periodic tasks. The formal semantics of THMSC is defined using labelled partially ordered set(*lposets*).

6.2.2 Time-Enriched LSCs

Harel and Marely [HM02] extend LSCs with time. The authors adopt (1) the approach presented by Alur and Henzinger [AH97], according to which a real-time system can be viewed as a discrete system with clock variables and (2) the synchrony abstraction hypothesis according to which system events consume no real time and time may pass only between events. A single clock object with one property, *Time*, and one operation/method, *Tick*, are used in combination with assignments and conditions to define timing constraints. Time can be stored in time-variables (i.e. Time-Variable := Time) and compared with time values (for instance the condition: *Time* $>$ *Time-Variable* $+$ *Min-Delay* is used to specify relative minimal delay whereas whereas *Time* $<$ *Time-Variable* $+$ *Max-Delay* is used to specify relative maximal delay). The authors distinguish three basic timing constraints:

- *Vertical Delay*: In a single object instance, time is stored upon the occurence of an event, then the following event is bound by two *hot* conditions defining the minimum and maximum delays of its occurence.
- *Message Delay*: A message delay is specified similarly, except that the time is stored in one instance line and is checked in another.
- *Timer*: A timer is also specified in the same manner as a vertical constraint, except that the maximal delay condition can be placed arbitrarily far from the place where the time is stored.

Conditions (*hot* and *cold*) can be used to combine timing constraints with conventional constraints to express complex timing constraints.

The timed LSC synchrony hypothesis (i.e. zero-time actions) is implemented in the play-engine simulation tool [HM01]. Indeed, while executing a timed LSC model, the clock keeps ticking and the system waits for external stimuli. When such a stimulus arrives, the execution freezes the clock and performs the sequence of events that constitutes the systems response to that stimulus. As the sequence is completed, the clocks operation is resumed. However, the authors [HM02] have mentioned that the synchrony assumption could be easily droped by letting the clock continue to tick when events and functions from the model are applied.

In another work by Klose and Wittke [KW01], LSCs are annotated by timers and by delay intervals (both a minimum and a maximum delay) expressing quantitative local liveness properties. However, these intervals are limiting the timing constraints to pairs of events that are either on the same instance line or are connected by a message. The operational semantics of an LSC is defined in terms of a symbolic timed Büchi automaton with unique clocks serving for each constraint. The procedure of deriving an automaton from an LSC is called *unwinding* [KW01].

6.2.3 Timed Annotations in UML

In the following subsections, a survey of timing annotations in UML1.x and UML2.0 diagrams is presented.

UML 1.x

UML 1.x offers nine different diagram types for specifying both structure and behavior of a system. To support real-time modeling, UML 1.x included graphical representation for timing mark to denote event occurrence time, time expression that evaluates to an absolute or relative value of time, and timing constraint which is a semantic statement about the relative or absolute value of time [GBJ96]. However, these added timing constraints are not available in all UML diagrams of the same model and they are generally informal in nature. In UML1.x, there is no time model that describes the way time is progressing. In the following subsections, a survey of timing annotations in UML1.x diagrams is presented.

UML Timed Sequence Diagrams. UML sequence diagrams use the drawing rules of message arrows and timing markers to express timing constraints [GBJ96]. A horizontal message arrow indicates the simultaneous occurrence of the send and receive events of the message. A downward slanted message arrow, on the other hand, indicates a required delay between the send and receive events of the message. In addition, within each object outgoing message arrows can be drawn at a single point to indicate the simultaneous sending of a message. Timing markers, boolean expressions placed in braces and attached to the sequence diagram, can also be used to constrain particular events or the whole diagram. These labels (interpreted as time stamps) can be attached at the beginning and the end of a message arrow to specify the minimum or maximum time gap between two marked points in the diagram.

Firley et al. [FHD+99] extend UML timed sequence diagrams to express loops. The sequence of messages which is repeated several times is surrounded by a rectangle with the loop condition (i.e. LOOP N TIMES expr) placed at the top or at the bottom of the rectangle. The following convention is used to deal with different occurrences of a labelled event in loops: before a loop, a_{first} can be used in constraints to refer to the first occurrence time of an event with time stamp a in the loop. After a loop, a_{last} refers to the last occurrence of the tagged event in the loop. Within a loop, a_{next} denotes the time of the event occurrence in the following iteration. The resulting diagrams are translated into observers and implemented in UPPAAL [LPY97]. UPPAAL models are then instrumented to be composed with the observers allowing for formal verification using model checking. However, the presented construction only supports totally ordered sets of events.

In [HHRS03, HHRS05] a trace based denotational semantics for timed sequence diagrams is formalized, called the timed STAIRS semantics (Steps to Analyze Interactions with Refinement Semantics). A timed trace is a sequence built from three kinds of events: events for transmission, reception and consumption. Each of these events may have an associated timestamp. The authors [HHRS03, HHRS05] claim that these three types of events are introduced to express distinction between black-box and glass-box view of a system. Li and Lilius [LL99b] study timing consistency of both basic UML sequence diagrams and composed sequence diagrams. They showed that the problem of time consistency checking can be reduced to linear programming (i.e. by solving systems of linear inequalities).

UML Timed Activity Diagrams. Eshuis et al. [EW01, Esh02] have proposed a formal semantics of UML activity diagrams in terms of Clocked Transition Systems (CTS) [MP96] that is suitable for workflow modeling. The authors [EW01] proposed two special event labels, *when(texp)* and *after(texp)*, denoting an absolute and a relative temporal event respectively, where global clock *gc* measures the current time and *texp* is an integer expression counting time-units of the global clock and the local clocks. These events are attached to activity diagram transitions. The authors consider also periodic events, events that are not specified at a single point in time but at a sequence of points. These events are modeled with the *when(cond)* each period construct.

Guelfi and Mammar [GM05] extend UML activity diagrams with timing constraints. Timing constraints include a time duration attached to each activity diagram node and two types of temporal event expressions *After(t)* and *When(t)* similar to the ones proposed in [EW01, Esh02]. The authors propose a formal semantics of UML timed-activity diagrams by mapping them to a Clocked Transition System (CTS) [MP96] restricted to integer variables modeling discrete real time aspects. The resulting semantics are translated into PROMELA language for formal verification. One of the limitations of this approach is that external events are not considered.

Xuandong et al. [XMY$^+$01] extend UML activity diagrams by introducing timing constraints. They introduce a time interval *[a,b]* that can be attached to a state *s*. The times *a* and *b* are relative to the moment at which the activity state *s* starts. Assuming that s starts at time c, then s may complete only during the interval *[c+a, c+b]* and must complete at the time *c+b* at the latest (i.e. must proceed to the next activity state at the time *c+b* at the latest). The authors propose a timing analysis method based on linear programming for UML activity diagrams (containing no loop) and an integer time verification technique for checking more general activity diagrams.

UML profile TURTLE [ACLdSS04], which is discussed in Section 6.2.3, supports temporal operators in activity diagrams.

UML Timed Statecharts. Timed Statecharts, proposed by Kesten and Pnueli [KP91], extend the traditional statecharts [Har87] by specifying time limits for the execution of transitions. Their semantics are defined with reference to a dense time domain. Transitions are classified into immediate transitions and timed transitions. Immediate transitions are triggered by inputs, but abstract from time consumption at all. Whenever an immediate transition is enabled, it must be executed before time can proceed. Timed transitions do not depend on inputs. Therefore, they focus on the modeling of time consumption and they are associated with a time interval (l,u) providing a minimum and a maximum waiting time. The lower bound signifies the minimum time that must be spent in the current state before a transition can be taken, while the possibly infinite upper bound limits the time during which the transition must be taken, if it is to be taken at all. Kesten and Pnueli [KP91] propose a so called *weak time* semantics, i.e., transitions requiring no enabling event and with an associatd delay τ and timeout v may be performed (non deterministically) at any time between τ

and v. The concept of a *step* is associated with the execution of an immediate transition; a reaction to an event may occur several steps after its generation, but still within the same timestamp. This kind of semantics is based on the fact that every generated event *persists* until the time no longer flow. The time may flow only if all the enabled transitions by that event have been executed.

Peron and Maggiolo-Schettini [PMS94] considered a version of statecharts with real-time features such as delays and timeouts and by allowing communicated signals with durations. Though occurrencies of actions are related to a dense time domain (i.e., positive rationals), the behaviour of statecharts is forced to be discrete. They have also extended the standard notion of reaction by allowing sequences of transitions to be performed instantaneously. Later, the authors [MSP96] proposed an approach that assigns a precise duration to transitions instead of time interval [KP91], and enforced a strong time semantics which avoids enabled transitions from being arbitrarily delayed and required that a non-deterministic choice among transitions performance is done only if they can be really performed at the same time. Their idea is to increase the duration of transitions having null duration, and to decrease duration of transitions having nonnull duration, so that the time necessary to perform each chain of transitions remains unchanged.

UML Profiles. In addition to the many aspects of UML 1.x that have been criticized (e.g. the metamodel, the usability, the potentially inconsistent diagrams and views, the composition of models, and the insufficient support of error handling, etc.) [HR00, MLLG01], Berkenkotte [Ber03] identified the following four weaknesses related to real-time development:

1. The definition of hardware-software interdependencies: deployment diagrams are too imprecise as they do not provide information on the hardware (except the information that there is hardware at all).
2. The specification of timing constraints like deadlines and periods.
3. Communication structures: messages exchange can be specified in various ways (sequence diagrams, collaboration diagrams, etc.), but detailed information like periodicity and protocols cannot be given.
4. Task management policies: UML does not provide mechanisms to describe certain aspects of task management like priorities.

To address some of these weaknesses, UML 1.x has been combined with other techniques like ROOM [SGW94] (discussed in Section 6.2.5) or SDL [IT02a] (discussed in Section 6.2.4). UML Profiles represent also an alternative to address some of these shortcomings [GK06]. The following subsections list some of the existing UML profiles for real-time modeling:

UML profile for Schedulability, performance and Time (SPT)

UML profile for Schedulability, performance and Time (SPT profile) [OMG02] was requested and later adopted by OMG in 2002 to support real-time modeling. UML/SPT is a framework to model resource, time, concurrency schedulability and performance concepts and to support quantitative analysis of UML models.

SPT time domain model identifies the set of time-related concepts and semantics that are supported, directly or indirectly, by this profile. The time domain model is partitioned into the following separate but related groups of concepts [OMG02]:

- Concepts for modeling time and time values.
- Concepts for modeling events in time and time-related stimuli.
- Concepts for modeling timing mechanisms (clocks, timers). In SPT, clocks were implicitly bound to the physical time.
- Concepts for modeling timing services, such as those found in real-time operating systems.

The underlined concepts are grouped into a set packages as shown in Figure 6.4.

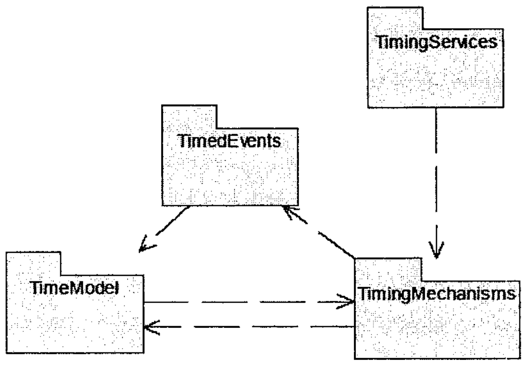


Figure 6.4: The Modules of the Time Domain Model [OMG02]

The sub-profile ≪*RTtimeModeling*≫ defines a metamodel representing time, as depicted in Figure 6.5, and time-related mechanisms, as illustrated in Figure 6.6, Figure 6.7 and Figure 6.8. The profile provides a set of stereotypes and associated tagged values that the modeler could apply to UML modeling elements:

- *TimeValue.* There are two ways to specify time values: (1) Use the *RTtime* stereotype to identify model elements that represent time values. The kind of time (discrete or dense) can be specified with an optional tag *RTkind*, which is an enumeration consisting of two elements: *dense* and *discrete.* (2) Use instances of the TVL data type RTtimeValue (or its subclasses), which is defined in this profile.
- *TimeInterval. RTinterval* stereotype is used to identify instance-based concepts that represent time intervals.
- *TimingMechanism.* The *RTtimingMechanism* stereotype is defined as an abstract stereotype that captures the common characteristics of timers and clocks.

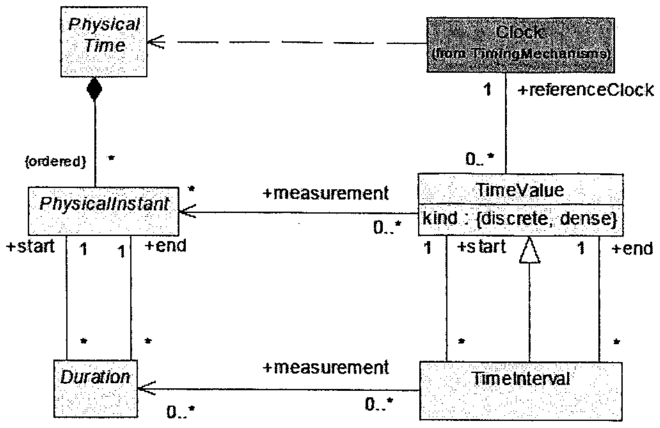


Figure 6.5: Time Modeling in UML/SPT [OMG02]

- *Clock.* They are modeled by applying the *RTclock* stereotype. An instance of a clock can be identified using the *RTclockId* tag.
- *Timer.* Timers are modeled by applying *RTtimer* stereotype.
- *TimedAction.* This concept is modeled by applying the *RTaction* stereotype to any model element that specifies an action execution or its specification. This includes action executions, methods, actions, action states, subactivity states, states, and transitions. It can also be applied to model stimuli that take time to arrive at their destination. The start and end times of the action are specified by appropriate tagged values (*RTstart* and *RTend* respectively). Alternatively, they may be tagged with the *RTduration* tag.
- *TimedEvent.* This concept is modelled by applying the *RTevent* stereotype to any model element that implies an event occurrence.
- *TimedStimulus.* This concept is useful for modeling any stimulus that has an associated timestamp. This includes invocations of operations, the sending of signals, etc. as well as their descriptors. The stereotype used for this purpose is the *RTstimulus* stereotype which can be attached to stimuli or action executions of actions that generate stimuli.
- *ClockInterrupt.* This is a special type of timed stimulus that is generated by a clock. The stereotype is called RTclkInterrupt and it can be applied either to stimuli or messages. The start time (*RTstart*) represents the time of the interrupt.
- *Timeout.* Timeouts are modeled by stimuli or messages that are stereotyped as *RTtimeout.* The start time *RTstart* represents the time of the timeout.
- *Delay.* This is modeled by a model element that is stereotyped as *RTdelay.* It can only have an *RTduration* tag associated with it. Delays can be placed on the same model elements as timed actions.

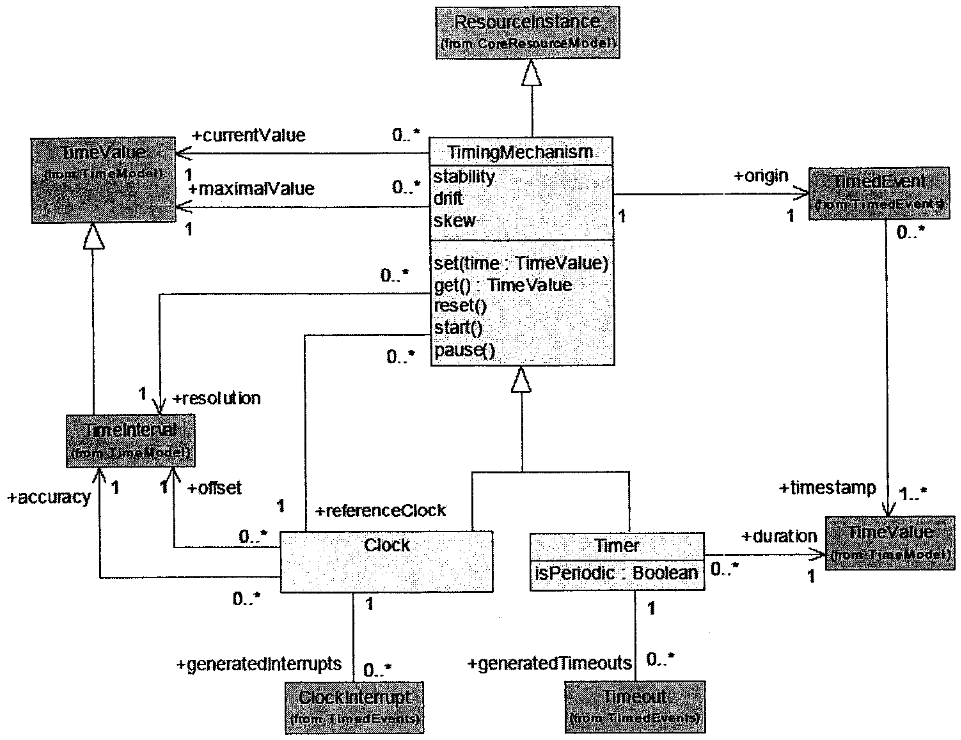


Figure 6.6: Timing Mechanisms In UML/SPT [OMG02]

- *TimeService.* This is represented by stereotype *RTtimeService.* Invocations of the operations of the time service are identified by corresponding stereotypes of ActionExecution or any model element that implies an action execution: *RTnewTimer* and *RTnewClock*.

Figure 6.9 illustrates an example of time annotations in sequence diagrams.

The SPT profile supports schedulability analysis of UML models by using ≪SAprofile≫. SPT Schedulability analysis may use modifiers on some parameters, such as: (1) worst-case values (as in, *worst-case execution time*), (2) special parameters of a task, such as its release time, its relative and absolute deadlines and laxity, and (3) special measures such as blocking time, pre-empted time. Woodside and Petriu [CP04a] address SPT Schedulability analysis capabilities and limitations. Other research attempts [KCH01, SKW00] integrate the schedulability theory with object-oriented real-time design.

In addition to SPT, OMG proposes another profile that supports the assessment of non-functional properties of software systems, called the Quality of Service and Fault Tolerance Characteristics and Mechanisms (QoS&FT) [OMG06] profile. *QoS&FT* allows the user to define a wider variety of QoS requirements and properties. However, QoS&FT requires a tremendous effort to be applied by the final users (software analyst, designer) [BP04]. For a comparative analysis between SPT and QoS&FT, the reader is invited to consult the work by Bernardi and Petriu [BP04].

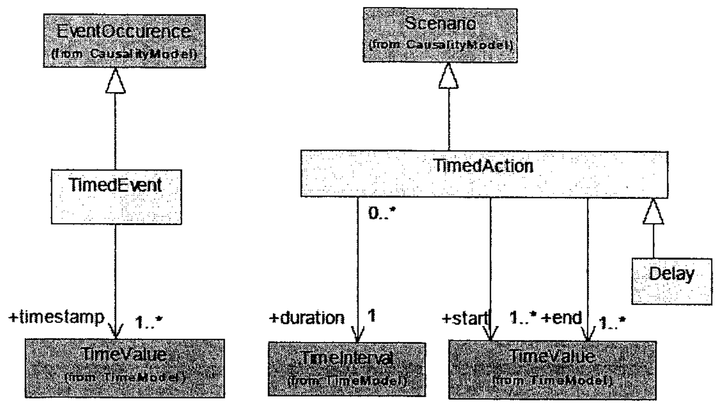


Figure 6.7: Timed Action and Timed Event Concepts [OMG02]

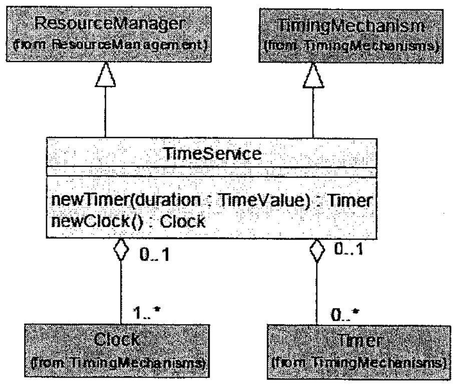


Figure 6.8: Timing Service Concepts [OMG02]

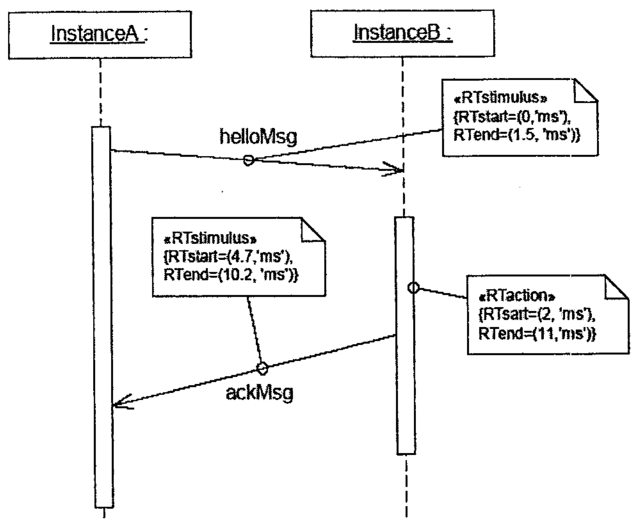

InstanceA :
InstanceB :
helloMsg
«RTstimulus» {RTstart=(0,'ms'), RTend=(1.5, 'ms')}
«RTstimulus» {RTstart=(4.7,'ms'), RTend=(10.2, 'ms')}
«RTaction» {RTsart=(2, 'ms'), RTend=(11,'ms')}
ackMsg

Figure 6.9: Time Annotations in Sequence Diagrams [OMG02]

Modeling and Analysis of Real-Time and Embedded Systems (MARTE)

The OMG has also recently issued a request for proposal (RFP) for a new UML profile for *Modeling and Analysis of Real-Time and Embedded Systems* (MARTE) [BM07b] in order to upgrade the SPT profile to UML2.0 [OMG05] and to extend its scope with real-time embedded system (RTES) modeling capabilities. MARTE goes beyond the SPT quantitative model of physical time and adopts more general time models. In MARTE, time can be *physical* (used by chronometric clocks), and considered as *dense* or *discretized*, but it can also be *logical* (i.e., bound to any recurrent event), which focus on the ordering of instants, possibly ignoring the physical duration between instants.

Espinoza et al. [EDG+05] provided a framework for MARTE by adopting the modeling practices of the SPT and QoS&FT, and proposed a domain model for annotating non-functional properties to support temporal verification of UML based models. Other UML profiles for different quantitative analyses have been proposed in the literature, such as reliability profile [CP04b] and dependability analysis profile [BM07a].

A UML Profile with the OCL

The Object Constraint Language (OCL) is part of the UML since version 1.3. In UML 1.x versions, OCL was used for specifying invariants attached to classes, pre- and post conditions of operation, and conditions on state transitions. However, it does not provide support for temporal constraints over the dynamic behavior of objects. It is not possible to reference different time instants in a single OCL formula. Only invariant properties can be formalized, which at most include references to attribute values before or after method execution. This lead to several OCL extensions [LMM05, Fla03, CK02, FM02a, FM02b] to address this limitation.

Lavazza et al. [LMM05] have proposed OTL (Object Temporal Logic) as a temporal logic extension to OCL. OTL provides the typical basic temporal operators of temporal logics, i.e., *Always*, *Sometimes*, *Until*, etc. In addition, OTL allows the modeler to reason about time in a quantitative fashion. OTL extends OCL 2.0 standard library by adding three new classes: *Time*, *Duration* and *Interval*. Class *Time* models time instants, which are defined based on the current time taken as the time origin. Class *Duration* models duration of time intervals, i.e., the distance between two time instants. Therefore, a time *Interval* can be defined by its initial time instant and its duration.

Cengarle and Knapp [CK02] extended OCL by satisfaction operators $@\eta$ to refer to the value in the history of an expression at the instant when event η occurred, as well as the modalities *always* and *sometime*. However, their approach deals with time only from a qualitative viewpoint where no notion of temporal distance between events is provided.

Flake and Mueller [FM02b] proposed a UML profile based on an extension of OCL 2.0 metamodel for the specification of real-time constraints. The formal semantics of this profile is given by means of a mapping to time-annotated temporal logic formulae expressed in CTL, which allows the formal verification of properties. The authors use a discrete time approach.

Sendall and Strohmeier [SS01] proposed an approach to specify concurrent operations through operation schema calculus based on OCL. They have introduced pre- and postcondition assertions, invariants, synchronization on shared resources, signals, and exceptions of system operations written in OCL. The authors have also introduced timing constraints on UML state machines in the context of a restricted form of UML protocol state machines called System Interface Protocol (SIP). A SIP defines the temporal ordering between operations. Five time-based attributes on state transitions are proposed, e.g., (absolute) completion time, duration time, or frequency of state transitions. In a related work, Marcel and de Boer [MdBFS04] define extension of OCL with a notion of event history that can be used for defining arbitrary constraints on such histories.

Non OMG Profiles

In addition to the aforementioned UML profiles, there are several *unofficial* proposals from the academia considering time modeling.

OMEGA-RT profile. This profile, part of the OMEGA project [OME07], aims to provide a concrete UML profile with formal semantics. It is a refinement of the SPT profile. It introduces events based time modeling, *TimedEvent*, where an event is used to represent an instant of state change and allows the expression of duration constraints between occurrences of events [GOO06]. OMEGA-RT profile defines a syntactic classification of events called *Event kinds*. For instance, in a signal exchange, three event kinds can be identified: *send*, *receivesignal* and *acceptsignal* events.

The profile is based on the existence of two basic types, *time* representing points in time *instants* and *duration* representing distances between time points. Sets of instants and durations are expressed by means of predicates. These predicates are formalizaed using OCL-like expressions. OMEGA-RT profile define the following duration patterns [GOO06]:

- execution time, execution delay, client response time, server response time, transmission delay which are associated with actions.

- reactivity and period which are associated with a trigger
- transmission delay associated with a communication channel
- lifetime associated with an object, and many more.

For requirements involving conditions, which are more complex than the distance between two events, OMEGA-RT introduce the *observer* formalism, defined by the stereotype class of state machine (≪observer≫). An observer is an object which executes synchronously with a system and monitors its state and the events that are occurring. Note that the OMEGA-RT event is different from the UML event, which poses a compliance issue.

TURTLE profile. TURTLE (Timed UML and RT-LOTOS Environment) [ACLdSS04] is a real-time UML1.5 [OMG03] compliant profile with formal semantics given in terms of RT-LOTOS. TURTLE profile extends class/object diagrams and activity diagrams of UML1.5 [OMG03]. TURTLE class diagram consists of *Tclasses* having special attributes called Gates. Gates are used by *TClass* instances, *TInstances*, to communicate and are specialized into *InGate* and *OutGate*. In addition, TURTLE introduces stereotypes called composition operators that are used to explicitly express parallelism, synchronization, and sequence relationships between *TClasses*. In TURTLE profile, activity diagrams implements the behaviour of a *TClass*. These activity diagrams use logical and temporal operators that allow expressing synchronization on gates with data exchange. For real time modeling TURTLE offers the following temporal operators: deterministic delay, nondeterministic delay, timelimited offer, and time capture operator(see Fig. 6.10). Time intervals are expressed by combining the deterministic and nondeterministic delays.

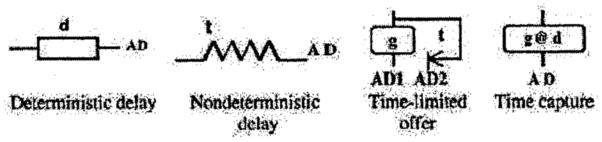


Figure 6.10: TURTLE Temporal Operators

TURTLE has been extended to include UML component and deployment diagrams. The resulting profile is called TURTLE-P [ALSS+06], which addresses the concrete description of communication architectures including quality of service parameters (delay, jitter, etc.). TURTLE-P allows the evaluation and formal validation of UML components and deployment diagrams. TURTLE is supported by TTool [TTo07]. TTool is linked to RTL [RL07b], the RT-LOTOS validation toolkit developped at CNRS, and to CADP [RL07a], a formal validation toolkit developed at INRIA.

EAST-EEA. The European EAST-EEA (Electronic Architecture and Software Technology – Embedded Electronic Architecture) [EE04] is an ITEA (Information Technology for European Advancement) project [ITE04]. It provides a development process and automotive-specific constructs for the design of embedded electronic applications. It provides the concepts of *Triggers*, *Period*, *Events*, *End to End Delay*, *Physical Unit* that can be applied to any behavioral EAST elements. In practice, some of these concepts, such as the event triggering, make the timing analysis very complex. In the EAST-ADL (Architecture Description Language) document, it is recommended to use event triggering carrefully or even to avoid it [AMPF07]. EAST-EEA is compliant with UML2.0.

Roubtsova et al. [RvKWR01] define a UML profile with stereotyped classes for dense time as well as parameterized specification templates for deadlines, counters, and state sequences. Each of these templates has a structural-equivalent dense-time temporal logics formula in Timed Computation Tree Logic (TCTL). The authors [RvKWR01], though, do not use OCL for constraint specification in their formal approach.

IF language

The IF language [BFG$^+$99, BFG$^+$00] and the associated toolset developed at VERIMAG were developed for modeling and validating distributed systems that can manipulate complex data and both involve dynamic aspects and real time constraints. The IF language describes the operational semantics of higher level formalisms such as UML or SDL, and is also used as a format for interconnecting model-based tools. An IF description defines the structure of a system and the behavior of its components. A system is composed of a set of communicating processes that run in parallel. IF provides support for real time constraints expressed using clock variables and guard conditions on them. The values of such variables increases with time. The underlying semantics is based on finite timed automata with urgency [AD94, BS00]. The IF language and tool-set [OGO06] translates timed UML models into timed automata in which UML level concepts are mapped into more primitive concepts. IF language format is used for the mapping and existing model-checking tools can be used for validation.

UML 2.0

UML 2.0 [OMG05] provides two data types: *Time* and *TimeExpression* to express timing constraints. It includes also time related concepts such as *timer* and *clock*. These timing statements can be used either in state diagrams or sequence diagrams as described in the previous sections. Moreover, UML 2.0 [OMG05] introduces a new diagram called *Timing Diagram* to allow reasoning about time and visualize conditions or state changes over time. Figure 6.11 illustrates an example of a Timing Diagram.

6.2.4 Time in SDL

The standard semantics of SDL, as presented in Z.100 [IT02a] (expressed by means of Abstract State Machines), is very abtract in the sense that it makes no assumptions on time consumption and progress. Tasks may take an indeterminate amount of time to execute, and a process may stay an indeterminate amount of time in a certain state before taking the next firable transition. Control over these durations is left to particular implementations by tool vendor (depends on application domain, implementation architecture, or purpose of simulation). However, SDL has some features

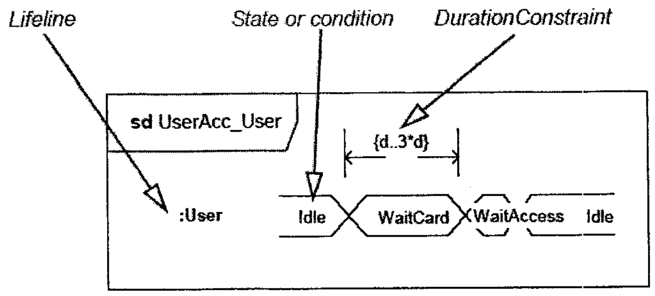


Figure 6.11: Timing Diagram Example [OMG05]

that can be used to model aspects of timed systems, such as global system time (represented by a system clock (*now*) and allows to measure durations throughout the system by means of appropriate time stamps). SDL supports two time-related data types: *Time* and *Duration*; *Time* should be used to denote a point in time, while *Duration* should be used to denote a time interval. System clock (*now*) is external to the specification. For example the system clock cannot be reset within the specification, nor does it progress in an orderly fashion. Rather, the only means for any form of control over the system clock is through the usage of timers.

SDL allows the description of time dependent functional behaviours by means of *timers*, *enabling conditions* and *continuous signals*. A timer expires when some delay has been exceeded, resulting in an input signal being placed in the input queue of the associated process. However, there is no guarantee when the signal will be consumed [BGM$^+$01]. SDL timer primitives are *set* and *reset* operations, the active funtion (which gives the state of a timer) and timeouts that are always transmitted in the form of asynchronous signals. An enabling condition, referring to the system time *now*, can be attached to an input signal. A continuous signal can also refer to *now*, where the intention is that when some time constraint is satisfied in a state, the behaviour of the process can evolve without environmental interaction (i.e. without an input signal). However, these two constructs do not allow the specification of transitions which are taken at a specific point of time (or within a specific time interval), as there is no notion of urgency in SDL. The current SDL semantics [IT02a] treats all transitions as lazy since it places no constraints on time progress. Most SDL tools however implement an eager semantics where transitions are fired as soon as they are enabled without letting time progress. Part of the European IST project INTERVAL (1999-2002), Bozga et al. [BGM$^+$01] propose a more flexible time semantics for SDL based on timed automata with urgencies [BST98].

Real-time distributed systems lack the notion of a global system clock, and thus global time. Graf [Gra02] proposed to introduce the notion of local time (defined by a drift and/or offset with respect to the global time *now*). She suggested that a defined relationship between the external reference time (*now*) and local time must always exist.

The QSDL [DHHMC95] defines an extension of SDL with probabilistic execution time constraints attached with tasks and a notation for some minimal deployment information. QSDL defines time durations (deterministic and probabilistic), timed transitions (using an action called *request*) and timed states (using the awake-construct). The resulting models are then fed into the tool QUEST, which transforms a QSDL-specification into an automatically assessable model (called an evaluator) for performance and time related verification.

6.2.5 Real-time Object Oriented Modeling (ROOM)

ROOM (Real-Time Object-Oriented Modeling), originally introduced in [SGW94], is a methodology that was developed primarily for distributed real-time systems based on the object paradigm. Modeling of systems with ROOM is performed by modeling actors (the central component of ROOM), which are encapsulated concurrent objects, communicating via point-to-point links. The behavior of an actor is represented by an extended state machine called a *ROOMChart*, based on Harel statechart formalism [Har87].Inter-actor communication is performed exclusively by sending and receiving messages via interface objects called *ports*. A message is a tuple consisting of a signal name, a message body (i.e., data associated with the message), and an associated message priority. The original ROOM [SGW94] does not provide any mechanisms to constrain the behaviour of actors (for instance to specify and enforce timing constraints). Instead system behaviour may be derived using Message Sequence Charts, which can be annotated with timing constraints [BAL97, SFR97]. The ROOM developers use the term *transaction*, to describe end-to-end computations on which timing constraints such as periodicity and deadlines may be specified. MSCs are used to express: (1) activation periods of methods (which represents either the inter-arrival time of the periodic timer, or a minimum inter-arrival time for aperiodically triggered transactions) and (2) end-to-end deadlines on sequences of message invocations (which represents the response time of the transaction). Using these two types of timing constraints and a few design guidelines, the authors in [SFR97] show how scheduling theory can be applied to ROOM models.

ROOM concepts were supported by a commercial CASE tool called *ObjecTime*(ObjecTime Ltd., Kanata, Ontario, Canada). They have been also incorporated into the CASE tool *Rational Rose Real-Time (RoseRT)* in the form of a UML profile, commonly called *UML-RT*.

6.2.6 Visual Timed Event Scenarios(VTS)

Alfonso et al. [ABKO04] introduced VTS, a visual language to define event-based requirements such as freshness, bounded response, event correlation, etc. The underlying language is based on partial orders and supports real-time constraints in a dense time domain. Figure 6.12 summarizes the VTS graphical notation.

The authors [ABKO04] provided a declarative (denotational) semantics of VTS, where a set of traces are assigned to each VTS scenario and labelled points represent events in the traces. Points that are not labelled are called *instants*. They represent moments in the execution not necessarily associated with an event. The resulting semantics are not executable. VTS is supported by a tool that translates visually specified scenarios (the ones violating the requirements) into observer timed automata. The resulting automata can be composed with a model under analysis in order to check satisfaction of the stated scenarios. However, describing graphically all possible scenarios

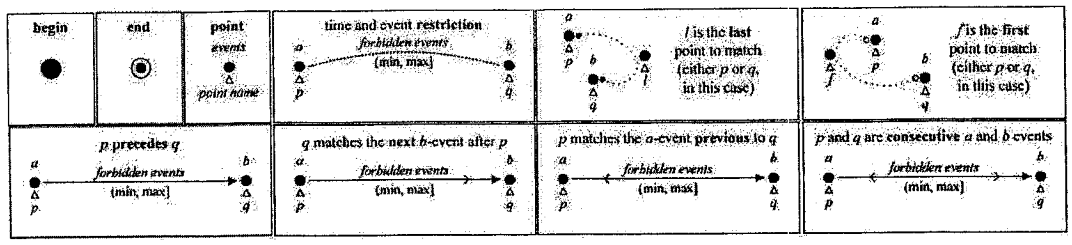

begin
end
point
events
point name
time and event restriction
forbidden events
[min, max]
l is the last point to match (either p or q, in this case)
f is the first point to match (either p or q, in this case)
p precedes q
q matches the next b-event after p
p matches the a-event previous to q
p and q are consecutive a and b events

Figure 6.12: VTS Graphical Notation [ABKO04]

that violate a given requirement is an error prone activity and the resulting set of scenarios may be incomplete.

Unlike other timed notations, such as LSC and timed sequence charts, VTS does not use the *timer* concept and it abstracts from the instances that perform events (e.g. message exchange).

6.2.7 Property Sequence Chart (PSC)

Property Sequence Chart (PSC) [AIP06] is a scenario-based visual language that extends the graphical notation of a subset of the UML 2.0 Interaction Sequence Diagrams. The authors in [AIP06] provide a comparison between PSC, UML 2.0 Interaction Sequence Diagrams and MSC, based on the existence of the following features: undesired/mandatory/provisional message, strict/weak sequencing, restrictions on intraMSGs, all of messages but one, simultaneous messages, interaction construct, parallel operator.

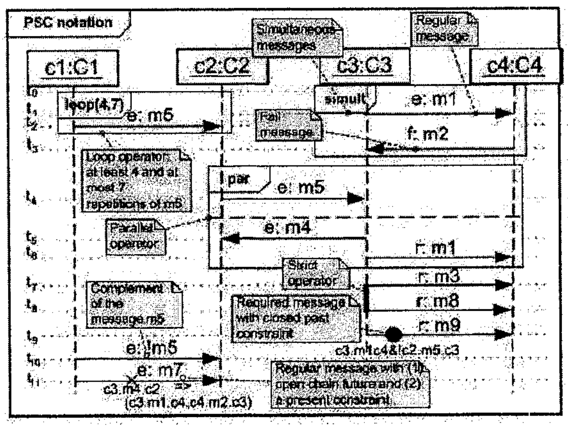

PSC notation
Simultaneous messages
Regular message
c1:C1
c2:C2
c3:C3
c4:C4
loop(4,7)
e: m6
simult
e: m1
Fail message
f: m2
Loop operator: at least 4 and at most 7 repetitions of m6
par
e: m5
e: m4
Parallel operator
r: m1
Strict operator
r: m3
r: m8
r: m9
Complement of the message m5
Required message with closed past constraint
e: !m5
c3.m1c4&!c2.m5.c3
e: m7
c3.m4.c2 => (c3.m1.c4,c4.m2.c3)
Regular message with (1) open chain future and (2) a present constraint

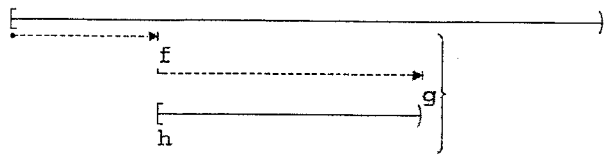
Figure 6.13: Property Sequence Chart [AIP06]

Although PSC provides a simple and user friendly formalism for specifying temporal properties, time support remains weak since the language does not allow the description of timing constraints. As shown in Figure 6.13, the only offered time notation consists of a set of horizontal dotted lines

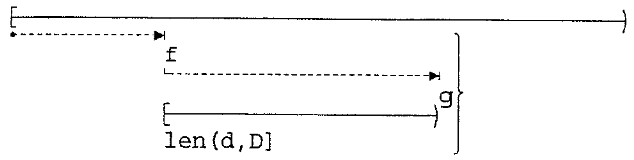
$t_0, \ldots, t_n$ called *time-lines*. No timing constraints are defined in order to be able to define a lower and an upper bound between two subsequent messages on one instance. The language is supported by a tool, called *CHARMY*, that can be used to translate specified properties into a test automaton (i.e., Büchi automaton).

6.2.8 Real-time Graphical Interval Logic (RTGIL)

Real-time graphical interval logic (*RTGIL*) [MRK+97], and its corresponding textual representation, real-time future interval logic (*RTFIL*), are real-time extensions to graphical interval logic (*GIL*) [RMSM+96], and its textual representation, future interval logic (*FIL*), respectively. *RTGIL* is a propositional linear-time temporal logic, interpreted over dense time. In *RTGIL*, a time line is used to show the progression of a computation. Intervals can be constructed on this time line; an interval is represented by a segment of the time line delimited by two states and is left-closed and right-open. Intervals are constructed using search patterns with associated target formulae. A search locates the first state in the future from the current position on the time line where the target formula holds (which might be the current state if the formula holds there). Formulae are read from top to bottom and from left to right, and can be combined using standard logical infix operators. Initial properties (Figure 6.14(a)) as well as henceforth or eventuality properties can be assigned to an interval. Figure 6.14(a) asserts that h holds at the first state if the interval that begins with the first state at which f holds and ends just prior to the next state at which g holds. The only real-time operator supported by *RTGIL* is the *len* predicate, for example (*len(d, D]* in Figure 6.14(b) implies that the duration of the interval, if it can be constructed, is greater than d time units and less than or equal to D time units (d and D represent non negative rational constants, where D can also be ∞).



(a) RTGIL Initial Property



(b) RTGIL Duration Constraint

Figure 6.14: RTGIL Examples [MRK+97]

RTGIL is supported by the RTGIL environment [MRK+97], which includes a graphical editor, an automated theorem prover, and a data base and proof manager component. Because the RTGIL environment is a homogeneous analysis tool, a model and its correctness properties are both specified in terms of RTGIL formulae.

6.2.9 Timeline Notation

A timeline [SHE01] is represented by a wide horizontal bar, with time progressing from left to right. Descending from the timeline bar are vertical bars, called *marks*, which mark the interesting event occurrences, ordered in time. Timeline defines two types of system events: regular events (denoted by the letter e) and required events (denoted by the letter r). The events can be generated anywhere in the system, by any one of many concurrent processes in the distributed system. Therefore, no fixed time-interval can be assumed between subsequent marks (there is no hidden assumption of a *global clock*). Timeline notation allows for describing constraints represented as black horizontal lines positioned beneath the timeline bar. These constraints are used to specify the occurrence of particular events over certain intervals. Figure 6.15 describes the fact that when the system must respond to an offhook by providing dialtone, the constraint !onhook must be placed within the interval between the ofhook and the dialtone event.

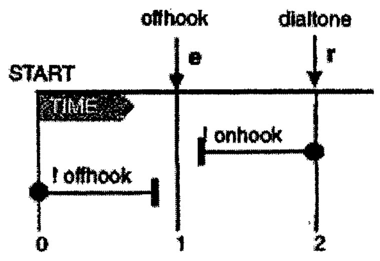


Figure 6.15: Timeline example [SHE01]

Timeline notation is supported by a graphical tool called TimeEdt [SHE01]. It was developed to capture series of events and required system responses. These complex chains are placed on a timeline and may be converted into a test automaton, that can be used directly by a logic model checker, or for traditional test-sequence generation. Even if intuitive, TimeEdt do not feature partial ordering of events and does not support complex timing constraints.

6.2.10 Regular Timing Diagrams (RTD)

Regular Timing Diagrams [AEN99, AEKN01] are a known notation in the context of hardware design. RTD diagrams describe, over a finite time period, changes of signal values, and precedence and timing dependencies between such events, such as *signal a rises within 5 time units of signal b falling* and *signal b is low when signal a rises.* Such events can be causally constrained and time-constrained by a number of ticks of a given clock where the time intervals are specified by constants, ensuring that the diagram defines a regular language.

A RTD may be either asynchronous or synchronous. A synchronous diagram (SRTD) includes one or more clocks with fixed periods and ensures that the time interval between any pair of events

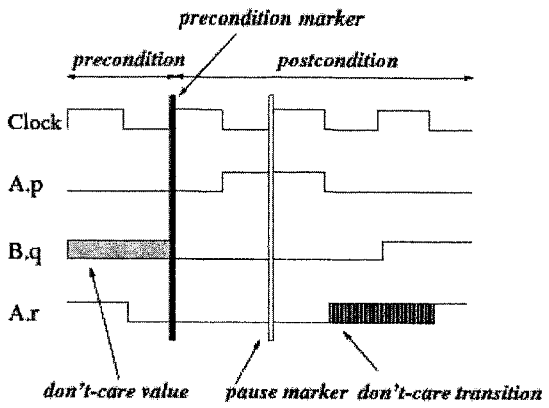


Figure 6.16: Synchronous Regular Timing Diagram [AEKN01]

is determined up to the clock period (see Figure 6.16). Any change in the signal value must occur at either the rising edge or falling edge of the clock waveform (which is between 0 and 1). The ordering between events is in general partial; such RTDs are considered as ambiguous. An unambiguous RTD has a total ordering on events.

6.2.11 Action Diagrams (Timing Diagrams)

An action diagram (AD) [Kho96, KC98] specifies in a declarative manner the interface behavior of a system. The specification comprises the interface behavior of the system itself (its commitments), as well as the assumptions that the system makes on its environment. Both commitments and assumptions are described by ports, actions, and timing constraints. Ports are abstractions of the logic signals used by the system to communicate with its environment. A direction (in or out) and a sequence of actions is associated with every port. Actions occur instantaneously; they represent punctual changes of the logic values of these signals. An action a_k has a time stamp variable denoted by $\mathrm{t}(a_k)$ which is a finite real value (dense time model).

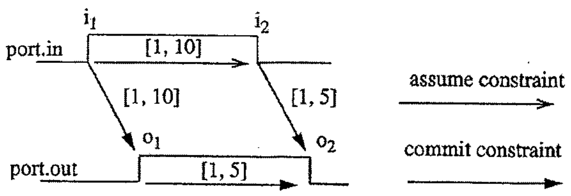


Figure 6.17: Example of Action Diagram [KC98]

In the graphical representation of action diagrams, an action is represented by a short vertical bar (e.g., Figure 6.17). Actions on the same port are horizontally aligned. The action sequence of a port is shown in left-to-right order. A constraint $(ai,\ aj,\ \pi)$ is represented by an arrow labeled with the interval π and pointing from a_i to a_j. The constraint arrowhead is hollow (filled) for assume (commit) constraints.

6.2.12 Timed Behavior Trees

In a recent work, Lars et al. [LKR07] extended the Behaviour Trees (BT) notation [Dro03] to include timing constraints. A timed BT model is equipped with a number of clocks which evaluate to a real number. All clocks progress simultaneously. A clock can be reset to zero or can constitute a guard on a transition or an invariant on a location.

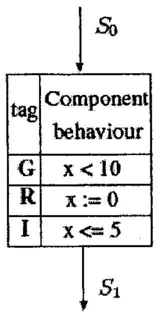


Figure 6.18: Timed BT Node [LKR07]

BT nodes [Dro03], as introduced in chapter 3, are extended with three additional slots (see Figure 6.18): a guard G over clock values, a reset R of clocks, an invariant I over clocks. Nodes in a timed Behavior Tree describe transitions from one location to the next as they describe a state change, a guard or message passing. The semantics of timed BTs are defined through a mapping to timed automata where each language concept has an equivalent automaton or a network of automata. The resulting semantics can be used as an input for the model checker UPPAAL.

6.2.13 Somé's Scenarios.

Somé et al. [SDV95, SDV96] represent timed scenarios with structured text, but also with a formal interpretation where preconditions, triggers, sequence of actions, reactions and delays are specified [SDV96]. Scenarios are interpreted as timed sequences of events, which make them appropriate for real-time systems. External events represent interactions between components, including actors, whereas actions can be internal. The time of occurrence of operations can be constrained by interaction *initial delays* and *timeouts* and scenario *timeouts*. An interaction *initial delay* specifies a minimal, a maximal or an exact amount of time that must pass between the interaction first operation, and the last operation of the interaction preceding it. Specifications described in this notation can be implemented in the Use Case Editor tool (UCEd)[Som04] and can be translated into a timed automata specification.

6.3 Summary of Evaluation of the Selected Timed Scenarios Languages

Tables 6.1, 6.2, 6.3, 6.4, 6.5 and 6.6 summarize the timed features of the thirteen timed scenario languages presented in the previous section. We use the following scale:

- −−: Absence of the feature: The language does not support the feature.

- -: Weak to basic existence: The language has a very basic support of the feature.

- +: Rich set of features: The language provides a fair/rich support of the feature.

- ?: Not specified: The notation does not specify the criteria.

- **N/A**: Not applicable: The feature does not apply to the notation.

Table 6.1: Survey Summary (1)

	Action/Event Enabling	Durational/ Instantaneous Events/Actions	Absolute/ Relative Time	Clocks	Urgency	Time Domain
Time Variants of MSC:						
ITU MSC-96 [IT96]	delay interval	instantaneous	relative	global	--	?
ITU MSC-2004 [IT04]	delay interval	instantaneous	relative/ absolute	global	--	dense/discrete
Meng-Siew [MS93]	delay interval	instantaneous	relative	global	--	?
Alur et al. [AHP96]	delay interval	instantaneous	relative	global	--	?
Li and Lilius [LL99a]	delay interval	instantaneous	relative	global	--	discrete
Ben-Abdallah and Leue [BAL97]	delay interval	instantaneous	relative	global	--	?
Grabowski et al. [GDO98]	delay interval	instantaneous	relative	global	--	?
Maigat and Hélouët [LMH00]	delay interval	instantaneous	relative	?	--	?
Zheng et al. [ZKH02]/ Zheng and Khendek [ZK02]	delay interval	instantaneous	?	?	--	?
Kim et al. [KC06] (THMSC)	delay interval	instantaneous	relative	global	--	dense/discrete
Time-Enriched LSCs:						
Harel and Marelly [HM02]	delay interval	instantaneous	absolute	local	--	discrete
Klose and Wittke [KW01]	delay interval	instantaneous	absolute	local	--	discrete
UML Timed Sequence Diagrams:						
Booch et al. [GBJ96]	delay interval	instantaneous	relative	?	--	discrete
Firley et al. [FHD$^+$99]	delay interval	instantaneous	relative	?	--	discrete
Haugen et al. [HHRS03] [HHRS05]	delay interval	instantaneous	relative	local	--	
Li and Lilius [LL99b]	delay interval	instantaneous	relative	global	--	discrete
UML Timed Activity Diagrams:						
Eshuis and Wieringa [EW01]	?	durational	absolute/ relative	global/ local	--	dense
Guelfi and Mammar [GM05]	?	durational	absolute/ relative	global/ local	--	dense
Xuandong et al. [XMY$^+$01]	?	durational	relative	global	--	discrete

Table 6.2: Survey Summary (2)

	Action/Event Enabling	Durational/ Instantaneous Events/Actions	Absolute/ Relative Time	Clocks	Urgency	Time Domain
UML Timed Statecharts:						
Kesten and Pnueli [KP91]	initiation and termination of enabling	durational	relative	local	+	dense
Peron and Maggiolo-Schettini [PMS94, MSP96]	delay interval	durational	relative	local	+	dense
UML Profiles:						
SPT [OMG02]	?	instantaneous/ durational	relative	local, physical	+	dense/discrete
MARTE [BM07b]	?	?	relative	local, physical/logical	+	dense/discrete
OCL:						
Lavazza et al. (OTL)[LMM05]	?	instantaneous	absolute	global	−−	dense/discrete
Cengarle and Knapp [CK02]	?	instantaneous	absolute	global	−−	dense/discrete
Flake and Mueller [FM02b])	?	?	absolute	?	−−	discrete
Sendall and Strohmeier [SS01]	?	?	absolute	?	−−	dense
non OMG Profiles:						
OMEGA-RT [OME07]	?	instantaneous	relative	local, physical	+	dense/discrete
TURTLE [ACLdSS04] and TURTLE-P [ALSS+06]	?	instantaneous	relative	local	+	dense/discrete
EAST-EEA [EE04]	?	?	?	physical	−−	?
Roubtsova et al. [RvKWR01]	?	?	?	?	−−	dense
IF language [BFG+99] [BFG+00]	?	instantaneous	relative	local	+ (TA)	dense

Table 6.3: Survey Summary (3)

	Action/Event Enabling	Durational/ Instantaneous Events/Actions	Absolute/ Relative Time	Clocks	Urgency	Time Domain
UML 2.0 Timing Diagram [OMG05]	?	Durational	absolute/relative	global	−−	dense/discrete /value chain
SDL:						
SDL [IT02a]	simple	N/A	absolute	physical, global	-	dense
Graf [Gra02]	iniation and termination of enabling	instantaneous	absolute	local	+	dense
QSDL [DHHMC95]	iniation and termination of enabling	durational	absolute	global	+	dense
Real-time Object Oriented Modeling(ROOM) [SGW94]	simple	instantaneous	relative	global	?	?
Visual Timed Event Scenarios (VTS) [ABKO04]	simple	instantaneous	relative	N/A	N/A	N/A
Property Sequence Chart (PSC) [AIP06]	simple	instantaneous	absolute	−−	−−	−−
Real-time Graphical Interval Logic (RTGIL) [MRK+97]	simple	instantaneous	relative	−−	N/A	dense
TimeEdt [SHE01]	simple	durational	absolute	−−	N/A	?
Regular Timing Diagrams (RTD) [AEN99]	simple	durational	absolute	global	−−	discrete
Action diagram (AD) [Kho96, KC98]	simple	instantaneous	relative	global	−−	dense
Timed Behavior Trees [LKR07]	iniation and termination of enabling	instantaneous	relative	local	+	dense
Some Scenarios [SDV95]	simple	instantaneous	absolute/ relative	?	−−	?

Table 6.4: Survey Summary (4)

	Time Representation/ Measurement	Time Constructs/ Constraints	Time Analysis	Spec Executability	Semantics
Time Variants of MSC:					
ITU MSC-96 [IT96]	point/interval	-	N/A	N/A	denotational
ITU MSC-2004 [IT04]	point/interval	+	N/A	N/A	event traces
Meng-Siew [MS93]	interval	-	+(bMSC only)	?	?
Alur et al. [AHP96]	interval	+	+(bMSC only)	+ (MSC Analyzer)	partial orders
Li and Lilius [LL99a]	interval	+	+(timing consistency)	?	local semantics
Ben-Abdallah and Leue [BAL97]	interval	+	+(temporal constraint graph)	+ (MSC tool)	event traces
Grabowski et al. [GDO98]	interval	-	?	?	constraint diagrams
Maigat and Héloutë [LMH00]	interval	-	+(max,+) automaton		partial order automata
Zheng et al. [ZKH02]/ Zheng and Khendek [ZK02]	point/ interval	+	+	-	timed Lposet
Kim et al. [KC06] (THMSC)	interval	+	+	?	timed Lposet
Time-Enriched LSCs:					
Harel and Marelly [HM02]	point/ interval	+	?	+(play engine)	-
Klose and Wittke [KW01]	point/ interval	+	?	?	timed büchi automata
UML Timed Sequence Diagrams:					
Booch et al. [GBJ96]	point/ interval	-	-	-	-
Firley et al. [FHD+99]	point/ interval	+	observers/ model checking	?	TA
Haugen et al. [HHRS03] [HHRS05]	interval	-(basic)	-	-	Timed STAIRS (denotational)
Li and Lilius [LL99b]	interval	+	+(timing consistency)	-	local semantics

Table 6.5: Survey Summary (5)

	Time Representation/ Measurement	Time Constructs/ Constraints	Time Analysis	Spec Executability	Semantics
UML Timed Activity Diagrams:					
Eshuis and Wieringa [EW01]	point	+	Model Checking (TCM tool)	-	CTS
Guelfi and Mammar [GM05]	point	+	+ Model Checking (PROMELA)	-	CTS
Xuandong et al. [XMY$^+$01]	interval	+	+(time consistency)	?	?
UML Timed Statecharts:					
Kesten and Pnueli [KP91]	interval	+	-	?	weak time semantics
Peron and Maggiolo-Schettini [PMS94, MSP96]	point/interval	+	-	?	strong time semantics
UML Profiles:					
SPT [OMG02]	point/interval	+	+ (WCET, resources, scheduling policies)	?	?
MARTE [BM07b]					
OCL:					
Lavazza et al. [LMM05]	point/interval	+ (OTL)	?	N/A	?
Cengarle and Knapp [CK02]	point	+ (OCL + temporal operators)	-	-	trace semantics
Flake and Mueller [FM02b])	interval	+ (OCL consistent)	Formal verification	N/A	Clocked CTL
Sendall and Strohmeier [SS01]	?	+ (OCL consistent)	-	N/A	-

Table 6.6: Survey Summary (6)

	Time Representation/ Measurement	Time Constructs/ Constraints	Time Analysis	Spec Executability	Semantics
non OMG Profiles:					
OMEGA-RT [OME07]	point/interval	+	+	?	
TURTLE [ACLdSS04] and TURTLE-P [ALSS+06]		+	+	+	RT-LOTOS
EAST-EEA [EE04]	point/interval	-	?	?	?
Roubtsova et al. [RvKWR01]	point/interval	+ (stereotyped classes)	?	?	TCTL
IF language [BFG+99] [BFG+00]	point/interval	+	+(model checking)	+	TA
UML 2.0 Timing Diagram [OMG05]	point/interval	-	+	+ (many tools)	?
SDL:					
SDL standard [IT02a]	interval	-	N/A	N/A	ASM
Graf [Gra02]	interval	+	?	?	Transition Urgencies
QSDL [DHHMC95]	point/ interval	+	+ (timed validation)	+ (QUEST)	?
Real-time Object Oriented Modeling(ROOM) [SGW94]	point/ interval	+(Schedulability analysis)	-	+ (UML-RT, object time)	EFSM
Visual Timed Event Scenarios (VTS) [ABKO04]	point/ interval	——	TA observers	——	-
Property Sequence Chart (PSC) [AIP06]	point	——	——	——	büchi automata
Real-time Graphical Interval Logic (RTGIL) [MRK+97]	point/ interval	+	threom prover	N/A	——
TimeEdt [SHE01]	-	-	——	-	-
Regular Timing Diagrams (RTD) [AEN99]	interval	——	——	——	——
Action diagram (AD) [Kho96, KC98]	interval	-	verification	+	operational
Timed Behavior Trees [LKR07]	point/ interval	+	+ (model checking)	?	TA
Some Scenarios [SDV95]	interval	+	+ (TA)	?	TA

6.4 Chapter Summary

The need to incorporate non-functional aspects, and in particular time-related aspects into requirement languages has been widely recognized. In this chapter, we have proposed a collection of eleven criteria that will help categorize and compare many timed scenario notations. These criteria are:

- Timed Action/Event Enabling
- Durational vs. Instantaneous Events/Actions
- Absolute vs. Relative Time
- Clocks: Local vs. Global and Physical vs. Logical
- Urgency
- Time Domain
- Time Representation and Measurement
- Time Constructs/Constraints
- Time Analysis and Verification
- Specification Executability and tool support
- Formal vs. Informal Semantics

Based on the proposed criteria, we have surveyed and compared thirteen timed scenario notations. The proposed criteria and classification represent a corner stone towards the extension of Use Case Maps language with time, presented in the next chapter.

Chapter 7

Timed Use Case Maps

UCM models focus on the description of functional requirements and high-level designs at early stages of the development process. Time based requirements can affect a system with respect to its correctness and performance. However timing issues are typically introduced later in the development process, which may result in considerable changes at the design or even worse at the requirements level. We believe that timing aspects must be integrated into the system model, and this must be done already at an early stage of development. The motivations for extending Use Case Maps language with time can be summarized as follows:

1. The existing UCM language does not describe semantics involving time, allowing for different interpretations of timing information, such as the time needed for a transition or a responsibility to complete.

2. The integration of timing aspects at an early stage of development allows for a consistent analysis throughout all lifecycle phases of software product.

3. Modeling of time semantics helps support further time related analysis (such as schedulability analysis) of UCM models.

4. Extending UCMs with time represents a first step towards the construction of a formal framework for using UCM to describe, simulate, analyze and verify real-time systems at high level of abstraction.

This chapter[1] introduces an approach to describe timing constraints in Use Case Maps specifications. First, an outline of the decision points related to time extensions is presented. Then, an extension of the syntax of UCM constructs with time is provided. The operational semantics of timed UCM are formalized in terms of Clocked Transition Systems (CTS) [MP96] (Section 7.3), Abstract State Machines (ASM) [Gur88] (Section 7.4) and Timed Automata (TA) [AD94] (Section 7.5). These three formalisms have different expressive power and tool support.

[1]This chapter content is published in System Analysis and Modeling: Language Profiles - SAM 2006 [HRD06]

7.1 Modeling Time in UCMs: Decision points

Considering the nature of UCM language and based on the set of criteria presented in Section 6.1, the following decision points are discussed:

1. **Timed responsibility enabling.** *Initiation and termination of enabling* [BG06] may represent a flexible and suitable choice for UCM abstraction level. Both a lower and upper bound may be imposed on the enabling of a responsibility. Four options may be considered (discrete time domain is used for illustration purpose only):

   (a) A responsibility R may be associated with a tuple (τ,τ'). Responsibility R is enabled (i.e. can start executing) τ units after the completion of its predecessor. This enabling is offered for τ' units and is retracted after.

   (b) A responsibility R may be associated with a tuple (τ,0). This type of enabling is called *punctual enabling*, where the enabling retracts if the responsibility is not taken immediately.

   (c) A responsibility R may be associated with a tuple (τ,$\perp$). This type of enabling is called *simple enabling*, where no upper bound is imposed on enabling. The responsibility is enabled τ units after its predecessor and never retracts. This may involve major (even infinite) system execution delays.

   (d) A responsibility R may be associated with a tuple (*minDL*,*maxDL*). This is a variant of the first option (a) but with an upper bound relative to the completion of the preceding construct. Responsibility R may be enabled any time between *minDL* and *maxDL* time units after the completion of its predecessor. In this case, *minDL* should be less or equal to *maxDL*. This option is equivalent to option (a) with *minDL*=τ and *maxDL* = τ + τ'.

   For simulation and validation purposes and in order to ensure a maximal progress semantics, only options a, b and d can be selected.

2. **Instantaneous (atomic) vs. durational actions.** Approaches that adopt instantaneous action semantics make the modeling more compact and easier to reason about. However, in the context of UCMs, it would be more suitable to consider durational semantics. Indeed, using durational semantics would help:

   - Realistically describe various system requirements ranging from real time system where actions take micro seconds to business process models where actions take days and even weeks.
   - Describe truly concurrent systems where at any given time t more than one action may be executing.

   In the context of Use Case Maps, time is only consumed by responsibilities. *minDur* and *maxDur* denote respectively the best and worst case execution time of a responsibility. Responsibility with a fixed duration have *minDur=maxDur=duration.* All other UCM constructs take one clock tick to complete.

3. **Relative vs. absolute time.** A timed constraint may be expressed using either an absolute time, where the time of occurrence of a responsibility refers to the execution starting time, or a relative time where the time of occurrence of a responsibility refers to the execution of a causally preceding responsibility. However, in the context of UCMs, relative time is preferred over absolute time because:

   - In an absolute time model context, changing the origin of time would impact all the constraints in the model.
   - In UCM models that contain loops, using absolute time would not be possible, because a responsibility is part of a loop and may be traversed multiple times with different time stamps. In addition, placing an absolute time constraint on a responsibility after a loop would constraint the number of times a loop is traversed. However, such information is only known at run-time. Figure 7.1 illustrates this situation.

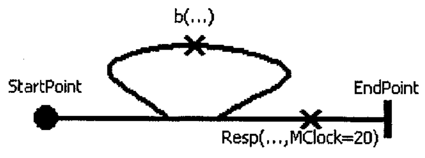


Figure 7.1: Absolute Time Constraint in Presence of UCM Loop

   A UCM model may have more than one start point. In such a case, an absolute time is required and the user may choose the time stamp of one start point to fix the origin of time, or have an independent origin. However, special attention should be given to such decisions, since it may impact the overall system constraints and behavior.

4. **Time representation and measurement.** Both *interval-based* and *point-based* representation can be used. An *interval-based* representation is used to estimate the execution time of a responsibility (i.e. [minDur, maxDur]) and to measure the execution time of an end-to-end scenario (e.g. latency measurement). Either a *point-based* or an *interval-based* representation can be associated to UCM *timestamps* (for instance MClock=10 or 1≤MClock ≤ 10).

5. **Dense vs. Discrete Time.** Apart from the complexity of reasoning in the verification and refinement area, using dense or discrete time will have minor effect on the proposed semantics.

6. **Global vs. Local Clocks.** A global and centralized clock for measuring and increasing time globally over the system is used (MasterClock (*MClock*) initially equal to zero). Local clocks are used to measure the time taken by a responsibility and in timers to set a duration, reset to zero and to observe for timeout. The smallest time unit used to track system evolution over time is named δ. It represents the clock tick and it also defines the granularity of the master clock.

7. **Urgency**. The concept of urgency is introduced into timed UCM semantics as follows:

- A responsibility R which is associated with a constraint (τ,τ') or (*minDL*,*maxDL*) (as introduced in options (a) and (d) of the first criteria) is considered as *urgent* when enabled immediately after the execution of its predecessor ($\tau = \tau' = 0$) or (*minDL*= *maxDL* = 0). Alternatively, it is considered as *delayable* when a delay is introduced ($\tau' \neq 0$ or *maxDL* $\neq 0$).
- Except responsibilities, all UCM constructs (i.e. control constructs such as OR-Fork, OR-Join, AND-Fork, etc.) are considered as urgent once enabled.
- Transitions are urgent and instantaneous: Transitions are processed as soon as they are enabled allowing for a maximal progress. Therefore, transitions can be considered as *eager* according to the definition of urgency introduced in [BST98].

7.2 Syntax of Timed Use Case Maps

Based on the discussion points presented in the previous section, our original Use Case Maps syntax presented in Section 4.1 is extended with time as follows:

Definition 5 (Timed Use Case Maps) . *We assume that a timed UCM is denoted by an 8-tuple (D, H, λ, C, GVar, B_c, B_s, MClock) where:*

- *D is the timed UCM domain, composed of sets of typed elements. D= SP $\cup$ EP $\cup$ R $\cup$ AF $\cup$ AJ $\cup$ OF $\cup$ OJ $\cup$ TS $\cup$ Tm $\cup$ ST. Where SP, EP, R, AF, AJ, OF, OJ, TS, Tm and ST are respectively the sets of Start Points, End Points, Responsibilities, AND-Fork, AND-Join, OR-Fork, OR-Join, Time stamps, Timers and Stubs.*
- *H is the set of edges connecting UCM constructs to each other.*
- *λ is a transition relation defined as: λ=D$\times$H$\times$D.*
- *C is the set of components (C = $\emptyset$ for unbound UCM).*
- *GVar is the set of global variables.*
- *B_c is a component binding relation defined as B_c =D$\times$C. B_c specifies which element of D is associated with which component of C. B_c is empty for unbound UCM.*
- *B_s is a stub binding relation and is defined as B_s =ST$\times$IN/OUT$\times$SP/EP. B_s specifies how the start and end points of the plug-in map would be connected to the path segments going into or out of the stub.*
- *MClock is the system master clock.*

The modeling of time is added as an orthogonal feature to the untimed UCM syntax presented in Section [HRD05a]. The untimed syntax can be restored simply by removing all execution delays as well as durations of responsibilities. The signature of timed UCM constructs is defined as follows:

Definition 6 (Timed UCM Constructs)

- **Start Points** are of the form *SP(PreCondition-set, TriggerringEvent-set, SP-label, in, out, minDL, maxDL)*, where the parameter *PreConditions-set* is a list of conditions that must be satisfied in order for the scenario to be enabled (if no precondition is specified, then by default it is set to true). The parameter *TriggeringEvents-set* is a list that provides a set of events that can initiate the scenario along a path. The parameter *SP-label* denotes the label of the start point. A start point should not have an incoming edge except, when connected to an end point (called a waiting place) or an entry edge of a stub. The parameter $in \in H$ represents such incoming edge. The parameter $out \in H$ is the (unique) outgoing edge. *minDL* and *maxDL* are respectively an optional absolute time lower and upper bound delay. They may be used to introduce a delay in the presence of more than one start point. *minDL* and *maxDL* are expressed relatively to the master clock *MClock*.

- **End Points** are of the form *EP (PostCondition-set, ResultingEvent-set, EP-label, in, out)*, where the parameter *PostConditions-set* is a list of conditions that must be satisfied once the scenario is completed. The parameter *ResultingEvent-set* is a list that gives the set of events that result from the completion of the scenario path. The parameter *EP-label* denotes the label of the end point; the parameter $in \in H$ is the (unique) incoming edge. End points have no target edge, except when connected to a start point (i.e. a waiting place) or to an exit edge of a stub. In such a case, $out \in H$ represents such connection. End points are not delayed.

- **Responsibilities** are of the form *Resp (in, R-label, out, minDL, maxDL, minDur, maxDur)*, where $in \in H$ is the incoming edge, *R-label* is the activity to be executed, and $out \in H$ is the outgoing edge. *minDur* and *maxDur* are respectively the minimal and maximal time allowed for a responsibility to complete its execution. As stated in Section 7.1, *minDL* and *maxDL* represent respectively a lower and an upper bound imposed on the enabling of a responsibility.

- **OR-Forks** are of the form OR-Fork (in, $[Cond_i]_{i\leq n}$, $[out_i]_{i\leq n}$) where in denotes the incoming edge, $[Cond_i]_{i\leq n}$ is a finite sequence of Boolean expressions, and $[out_i]_{i\leq n}$ is a sequence of outgoing edges. Conditions may involve timed constraints such as 'MClock $\leq$ 5'.

- **OR-Joins** are of the form OR-Join ($\{in_i\}_{i\leq n}$, out) where $\{in_i\}_{i\leq n}$ denotes the incoming edges and, out is the outgoing edge.

- **AND-Forks** are of the form AND-Fork (in, $\{out_i\}_{i\leq n}$) where in denotes the incoming edge, and $\{out_i\}_{i\leq n}$ is a set of outgoing edges.

- **AND-Joins** are of the form AND-Join ($\{in_i\}_{i\leq n}$, out) where $\{in_i\}_{i\leq n}$ denotes the incoming edges, and out is the outgoing edge. Time elapses in AND-Join while waiting for all incoming edges to synchronize). Such delays are conditioned by the internal execution of the system and do not represent a user requirement.

  Note: OR-Fork, Or-Join, AND-Fork and AND-Join are executed without delay. No relevant user requirements may suggest such delays.

- **Timers** are of the form Timer (in, TriggerringEvent-set, cont_path, to_path, TO). The synchronous timer is similar to a basic OR-Fork with two outgoing disjoint branches. The parameter TriggeringEvents-set is the list that contains the set of events that can trigger the continuation path (i.e. cont_path) and the parameter *to_path* $\in H$ denotes the timeout path. TO is the timer's expiration time.

- **Stubs** have the form *Stub(*$\{entry_i\}_{i \leq n}$, $\{exit_j\}_{j \leq m}$, *isDynamic,* $[Cond_k]_{k \leq l}$, $[plugin_k]_{k \leq l}$*)* where $\{entry_i\}_{i \leq n}$ and $\{exit_j\}_{j \leq m}$ denote respectively the set of the stub entry and exit points. *isDynamic* indicates whether the stub is dynamic or static. Dynamic stubs may contain multiple plug-ins $[plugin_k]_{k \leq l}$ whose selection can be determined at run-time according to a selection-policy specified by the sequence of Boolean expressions $[Cond_k]_{k \leq l}$. The sequence Cond is empty for static stubs (i.e. isDynamic=false). No time constraints are defined for stubs since a stub is a simple container for plug-ins and the execution of a stub is the execution of the selected plug-in.

7.3 Formal Semantics of Timed UCM Models in Terms of CTS

In this section, we define the formal semantics of timed UCM models in terms of Clocked Transition Systems (CTS) [MP96]. The proposed semantics consider a discrete time model to be divided into clock ticks indexed by natural numbers. The elapsed time between the events is measured in terms of ticks of a global digital clock which is increased by δ with every single tick. This time model corresponds to the fictitious-clock model from [AD94] or the digital-clock model from [HMP92].

The following sub-section provides an introduction to Clocked Transition Systems (CTS) [MP96] formalism.

7.3.1 Clocked Transition Systems

A clocked transition system is a tuple Φ= (V, Θ, T, Π) that consists of the following components:

- **System Variables V:** V is a finite set of system variables. It is divided into two subsets - discrete variables, which can be of any type, and clock variables. One special clock variable is the master clock T. The possible values of the system variables are called states. We use the term assertion to refer to a first-order formula, whose free variables belong to V.

- **Initial Condition Θ:** Θ is an assertion, characterizing the possible initial states. The initial condition implies that the master clock T = 0.

- **Transitions T:** T is a finite set of transitions. Transitions $\tau \in$ T assign new values to the system variables. Each transition is described by the assertion ρ_τ, which is called transition relation. Transitions can be constrained by an enabling condition - an assertion that describes when a transition can take place. Transitions are discrete and happen instantaneously. That is why they are not allowed to modify the master clock T.

- **Time-Progress Condition** Π: Π is an assertion which specifies a global restriction on the passage of time.

7.3.2 CTS-based Semantics of Timed UCM

Definition 7 (Timed UCM'Clocked Transition System) *Formally a timed UCM CTS is defined as:* $\Phi = (V, \sigma_{init}, \rightarrow, \Pi)$ *consisting of:*

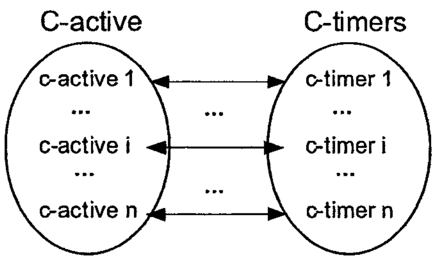
- V = *H-taken* $\cup$ *C-active* $\cup$ *H-enabled* $\cup$ *C-timers* $\cup$ *T-trigger* $\cup$ *MClock.*
  Where:
  - H-Taken *represents the set of already traversed edges. This set grows during the execution of a UCM model. It is specially useful in determining whether an AND-Join is triggered or not (all its incoming edges are already traversed).*
  - C-active *represents the UCM constructs currently executing. C-active is implemented as a sequence.*
  - H-enabled *represents the set of enabled edges (i.e. to be traversed during the next transition)*
  - C-timers *represents the remaining execution time (i.e., local clocks) of currently executing constructs contained in* C-active. C-timers *is initialized with the duration of execution of every construct in* C-active. *The duration of a responsibility is chosen randomly within interval [minDur, maxDur], while the duration of all other control constructs is assumed to be equal to one clock tick* δ.
  - T-trigger *represents the set of timers associated with the delay of the next UCM construct to be executed. For responsibilities, the delay is chosen randomly within [minDL, maxDL] interval. All control constructs should be executed as soon as they are triggered. Hence, once a UCM control construct is added to the set* C-active, *a value of one clock tick* δ *is associated to it in set* T-trigger. *This way, the control construct is executed during the subsequent clock tick.*
  - *MClock is the Master Clock.*

  The elements of sets T-trigger *and* C-timers *are decremented by* δ *in every transition. Values that reach zero are not decremented further.*

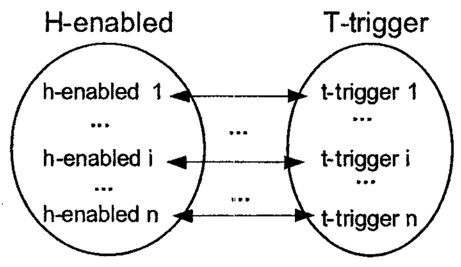
  Note that a bijection function is established between sequences C-active *and* C-timers *(see figure 7.2(a)), and between sequences* H-enabled *and* T-trigger *(see figure 7.2(b)). During a system transition, if an element is added to C-active (respectively removed from C-active), a corresponding element is added to C-timers (respectively removed from C-timers).*

- σ_{init}*: represents the initial state. It is required that for the initial state* MClock = 0.

- $\rightarrow$*: A finite set of transitions. Each transition is a function* $\rightarrow \subseteq \Sigma(V) \times \Sigma(V)$ *mapping each state* $s \in \Sigma$ *into a set of successors states* $s' \in \Sigma$. *Instead of writing* $(\sigma, \sigma') \in \rightarrow$, *we write* $\sigma \rightarrow \sigma'$.



(a) Bijection between C-active and C-timers



(b) Bijection between H-enabled and T-trigger

Figure 7.2: Bijections Functions within V

Informally, states are assignments of values to variables, called valuations. A valuation maps a variable to a value. A transition from one state to another represents that some variables are assigned a different value, i.e., the valuation changes.

- Π: *The master clock MClock is incremented by a clock tick δ at every transition. No time progress is allowed without executing a transition (configuration or time transition). Transition types will be discussed in Section 7.3.3.*

Definition 8 (Run) *A run of Φ is an infinite sequence of valuations, $\pi = \sigma_0\sigma_1\ldots$ satisfying:*

- *Initiation : $\sigma_0 \models \sigma_{init}$*
- *Consecution: For each i=0,1,... the valuation σ_{i+1} is a $\rightarrow$ successor of σ_i, i.e, $\sigma_i\rightarrow \sigma_{i+1}$.*

A computation of Φ is a run satisfying:

- *Time divergence: The sequence σ_0(MClock) σ_1(MClock)... grows beyond any bound. That is, as* i *increases, the value of MClock at σ_i increases beyond any bound.*

We assume that the run-to-completion principle applies to the execution of a construct. The execution of a UCM construct cannot be interrupted until it is completed.

7.3.3 CTS Transition Relation

In order to define the transition relation (i.e., $\rightarrow$), the following access functions are defined to access different elements of the proposed timed UCM data structures. These access functions abstract the transition relations from cumbersome details.

Definition 9 (Access functions)

1. **enables**: D $\rightarrow H^n$. Given a UCM construct, function *enables* provides the set of edges that the construct enables after it completes its execution (i.e., enables(Resp)={out}). Outgoing edges may be associated with guard conditions (i.e., OR-fork and dynamic stubs). Function *enables* evaluates the guards and chooses the outgoing edge associated with the true condition.

2. **incoming**: D$\rightarrow$ H^n. Given a UCM construct, *incoming* provides the set of edges directly leading to the construct.

3. **target**: H $\rightarrow$ D. Gives the subsequent construct directly connected to a given edge.

4. **delay**: D $\rightarrow \mathbb{N} \times \delta$. Computes the delay associated with a UCM construct. The delay is chosen randomly within [minDL, maxDL] interval. *delay= Random(Resp(..., minDL, maxDL)).*

5. **triggered**: $D^n \rightarrow D^m$, such that n$\geq$m. Given a set S of UCM constructs, *Triggered* produces the set S'$\subseteq$ S of UCM constructs that are triggered at the present time. A UCM construct is said to be triggered if and only if the following two conditions are met:

   - Its incoming edges are enabled.
   - No explicit delay is associated to it.

6. **duration**: D $\rightarrow \mathbb{N} \times \delta$. Gives the duration of execution of a construct. For a responsibility, the function returns a random value between *minDur* and *maxDur.* For all control constructs, it returns δ (one clock tick).

7. **type**: D$\rightarrow$\{SP, R, EP, AJ, AF, OJ, OF, Tm, Ts, ST\} specifies the type of a UCM construct.

We distinguish two types of transition relations:

1. **Configuration Transitions**: When a Configuration Transitions is taken, the system configuration defined by the three sets: *H-taken*, *C-active* and *H-enabled* is updated to indicate which transition has just been taken.

   (H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)$\rightarrow$(H$'$-taken, C$'$-active, H$'$-enabled, C$'$-timers, T$'$-trigger, MClock$'$).

   Where (H$'$-taken$\neq$H-taken) $\wedge$ (C$'$-active$\neq$C-active) $\wedge$ (H$'$-enabled$\neq$H-enabled) $\wedge$ (T$'$-trigger$\neq$T-trigger) $\wedge$ (C$'$-timers=C-timers - δ) $\wedge$ (MClock$'$=MClock + δ).

   In sections 7.3.5 and 7.3.6, we describe the rules that govern the update of the underlined sets.

   Generally, a configuration transition is executed upon the expiration of:

   - One of *C-timers* elements (i.e., $\exists$ c $\in$ C-timers such that, c $\leq \delta$). Hence, the system configuration is changed after one clock tick.
   - One of *T-trigger* elements (i.e., $\exists$ t $\in$ T-trigger such that, t $\leq \delta$). Hence, the system configuration is changed when the delay associated with a construct elapses.

   Note: In the rest of this chapter and for the sake of simplicity, the condition '$\exists$ c $\in$ C-timers such that c $\leq \delta$' is expressed by 'c $\leq \delta$' and the condition '$\exists$ t $\in$ T-trigger such that t $\leq \delta$' is expressed by 't $\leq \delta$'.

2. **Time Transitions**: When a time transition is taken, the system configuration remains unchanged.

   A time transition is executed, when one of the following conditions is met:

- One responsibility, part of *C-active*, is still executing (i.e., $\exists$ t $\in$ C-timers such that, t $>$ δ).
- One construct is delayed (i.e., $\exists$ t $\in$ T-trigger such that, t $>$ δ).

The global time *MClock* is incremented by a clock tick δ, while the set elements of C-timers and T-trigger are decremented by a clock tick δ.

(H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)$\rightarrow$(H$'$-taken, C$'$-active, H$'$-enabled, C$'$-timers, T$'$-trigger, MClock$'$)

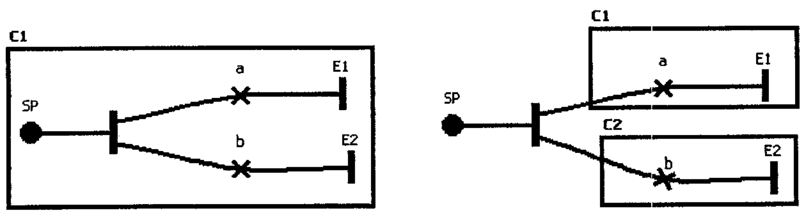
Where (H$'$-taken=H-taken) $\wedge$ (C$'$-active=C-active) $\wedge$ (H$'$-enabled = H-enabled) $\wedge$ (C$'$-timers=C-timer-δ) $\wedge$ (T$'$-trigger=T-trigger-δ) $\wedge$ (MClock$'$ = MClock + δ)

In sections 7.3.5 and 7.3.6, we describe the rules that govern the update of the underlined sets.

7.3.4 Concurrency Model and Time Evolution

The UCM construct AND-Fork allows many paths to execute concurrently. Considering the assumption of run to completion introduced earlier, different scenarios may behave either in:

- **Interleaving Semantics.** At any given time t, only one responsibility is currently executing. Or
- **True concurrency Semantics.** At any given time t, more than one responsibility is currently executing.



(a) Interleaving Semantics (b) True-Concurrency

Figure 7.3: Concurrency Semantics

We assume that in presence of UCM components, concurrent paths bounded to the same component are sharing also the same component resources. Therefore, these concurrent paths must behave in interleaving semantics. Figure 7.3(a) illustrates a UCM with two parallel paths bounded to one component. At any time, no more than one responsibility should be active. However, in this case, the choice of which responsibility goes first is non deterministic. Adding timing constraints may eliminate this non determinism (i.e., if responsibilities *a* and *b* have different values in *T-trigger*).

Parallel paths bounded to different components may behave either according to interleaving semantics or to true concurrency semantics. Figure 7.3(b) illustrates two parallel paths allocated to two different components. Responsibilities a and b can be executed in true-concurrency model, since they are enabled at the same time and they don't share the same resources. However, the decision to go with either semantics depends on the timing information that may be attached to these responsibilities.

Note: We assume interleaving concurrency model for unbound UCMs.

In what follows we provide the detailed semantic rules for both concurrency models. For the sake of simplicity, we consider only unfolded UCMs, where all stubs in the root map were already replaced with their corresponding plug-in maps.

7.3.5 Step Semantics for Interleaving Model

The choice of an interleaving semantics reduces the size of the CTS variables. Indeed, allowing only one construct to be executed in a given configuration, reduces the size of the used variables. Therefore, sets *C-active* and *C-timers* are reduced to singletons, since only one variable per set is necessary to track the configuration evolution.

As stated in the previous sections, a configuration transition is executed upon the expiration of one of the elements of either *C-timers* (i.e. a UCM construct finished executing) or *trigger*(i.e. a UCM construct should start executing). Alternatively, time transitions are taken.

Four possible conditions can be initially distinguished:

- Condition 1: $(t \leq \delta) \wedge (c \leq \delta)$. This condition triggers a configuration transition.
- Condition 2: $(t > \delta) \wedge (c \leq \delta)$. This condition triggers a configuration transition.
- Condition 3: $(t > \delta) \wedge (c > \delta)$. This condition triggers a time transition.
- Condition 4: $(t \leq \delta) \wedge (c > \delta)$. This condition triggers a time transition since the execution of a construct (e.g. execution of a responsibility) take precedence over configuration transitions. Hence, allowing for run to completion.

Since time may elapse in AND-Join constructs, where incoming flows should synchronize (time passes by while waiting for all incoming edges to be enabled), a special attention should be paid when an AND-Join is encountered. In what follows, different rules are devised to take care of AND-Join specificity.

Initial State: σ_{init} is defined with the following valuation: H-taken = $\emptyset$, C-active = $\emptyset$, H-enabled = incoming(StartPoints), T-trigger = delay(H-enabled), C-timers = $\emptyset$, MClock = 0.

In a given system state, the configuration transition (rule 1) is executed if all the following conditions hold:

(a) Condition 1 or Condition 2

(b) $\forall$ e $\in$ H-enabled, type(target(e))$\neq$AJ

$$\textbf{Rule 1}\quad \frac{\begin{array}{c} h:=\{\textit{any } e \in \textit{H-enabled, such that delay(e)=minimum(H-enabled) and delay(e)}\leq\delta\} \\ \textit{H}'\textit{-taken:=H-taken} \cup \{h\} \\ \textit{C}'\textit{-active:=triggered(target(h))} \\ \textit{H}'\textit{-enabled:=H-enabled} \cup \textit{enables(C-active)-}\{h\} \\ \textit{T}'\textit{-trigger:= (T-trigger-}\delta\textit{)} \cup \textit{delay(target(H}'\textit{-enabled))} \\ \textit{C}'\textit{-timers:= duration(C}'\textit{-active)} \\ \textit{MClock}'\textit{:=MClock} + \delta \end{array}}{\begin{array}{c}\textit{(H-taken, C-active, H-enabled, C-timers, T-trigger,MClock)}\rightarrow\textit{(H}'\textit{-taken, C}'\textit{-active, H}'\textit{-enabled,} \\ \textit{C}'\textit{-timers}'\textit{, T}'\textit{-trigger, MClock}'\textit{)}\end{array}}$$

Note: The edge h ∈ H-enabled, is non deterministically chosen amongst eligible edges.

In given system state, the configuration transition (rule 2) is executed if all the following conditions hold:

(a) Condition 1 or Condition 2

(b) ∃ e ∈ H-enabled, such that type(target(e))=AJ and incoming(target(e))$\subseteq$ H-enabled

$$\textbf{Rule 2}\quad \frac{\begin{array}{c} h:=\{\textit{all } e \in \textit{H-enabled, such that type(target(e))=AJ and incoming(target(e))}\subseteq \textit{H-enabled}\} \\ \textit{H}'\textit{-taken:=H-taken} \cup h \\ \textit{C}'\textit{-active:=triggered(target(h))} \\ \textit{H}'\textit{-enabled:=H-enabled} \cup \textit{enables(C-active)-}\{h\} \\ \textit{T}'\textit{-trigger:= (T-trigger-}\delta\textit{)} \cup \textit{delay(target(H}'\textit{-enabled))} \\ \textit{C}'\textit{-timers:= duration(C}'\textit{-active)} \\ \textit{MClock}'\textit{:=MClock} + \delta \end{array}}{\begin{array}{c}\textit{(H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)}\rightarrow\textit{(H}'\textit{-taken, C}'\textit{-active, H}'\textit{-enabled,} \\ \textit{C}'\textit{-timers, T}'\textit{-trigger, MClock}'\textit{)}\end{array}}$$

In given system state, rule 1 is executed if all the following conditions hold:

(a) Condition 1 or Condition 2

(b) ∃ e ∈ H-enabled, such that type(target(e))=AJ and incoming(target(e))$\subsetneq$ H-enabled

In given system state, the time transition (rule 3) is executed if all the following conditions hold:

(a) Condition 3 or Condition 4

(b) ∀ e ∈ H-enabled, type(target(e)) $\neq$ AJ

$$\textbf{Rule 3}\quad \frac{\begin{array}{c} \textit{C}'\textit{-timers:=C-timers-}\delta \\ \textit{T}'\textit{-trigger:=T-trigger-}\delta \\ \textit{MClock}'\textit{:=MClock} + \delta \end{array}}{\begin{array}{c}\textit{(H-taken, C-active, H-enabled, C-timers, T-trigger, MClock)}\rightarrow\textit{(H-taken, C-active, H-enabled,} \\ \textit{C}'\textit{-timers, T}'\textit{-trigger, MClock}'\textit{)}\end{array}}$$

Note: In time transitions, AND-Join is treated the same way as any other UCM construct.

Interleaving Model: Illustrative Examples:

This section illustrates the execution of configuration and time rules to produce system runs. For the sake of clarity, clock tick δ is chosen to be equal to 1 in the following timed UCM examples.

1. **Urgent responsibilities (no delay).** Figure 7.4 illustrates a simple timed UCM having an urgent (i.e., minDL = maxDL = 0) responsibility a with a duration interval [2,4]. In its corresponding run, described in table 7.1, the function *duration* assigns the value 3 to the duration of responsibility a.

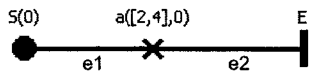


Figure 7.4: Urgent Responsibility

Transition type	$H - taken$	$C - active$	$H - enables$	$T - trigger$	$C - timers$	$MClock$
Initial State	$\{\}$	$\{\}$	$\{in1\}$	$\{\}$	$\{\}$	0
Configuration	$\{in1\}$	$\{S\}$	$\{\}$	$\{\}$	$\{1\}$	1
Configuration	$\{in1\}$	$\{\}$	$\{e1\}$	$\{0\}$	$\{\}$	2
Configuration	$\{in1, e1\}$	$\{a\}$	$\{\}$	$\{\}$	$\{3\}$	3
Time	$\{in1, e1\}$	$\{a\}$	$\{\}$	$\{\}$	$\{2\}$	4
Time	$\{in1, e1\}$	$\{a\}$	$\{\}$	$\{\}$	$\{1\}$	5
Configuration	$\{in1, e1\}$	$\{\}$	$\{e2\}$	$\{0\}$	$\{\}$	6
Configuration	$\{in1, e1, e2\}$	$\{E\}$	$\{\}$	$\{\}$	$\{1\}$	7
Configuration	$\{in1, e1, e2\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	8

Table 7.1: Urgent Responsibility Execution

2. **Delayed responsibility.** Figure 7.5 illustrates a simple timed UCM having a delayed (i.e., τ = 3) responsibility a with a duration interval [2,4]. Its associated run is described in table 7.2.

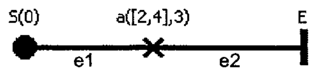


Figure 7.5: Delayed Responsibility

3. **Parallel Flows.** Figure 7.6 illustrates a simple timed UCM having two parallel flows. Its associated run is described in table 7.3.

4. **Synchronization of flows.** Figure 7.7 illustrates a simple timed UCM having two parallel flows synchronizing at an AND-Join. Its corresponding run is described in table 7.4.

7.3.6 Step Semantics for True Concurrency Model

Contrary to the interleaving semantics, sets *C-active* and *C-timers* may have more than one element in presence of concurrent paths. Indeed, *C-active* contains UCM constructs that are being executed currently and *C-timers* contain their respective set of timers (i.e. local clocks).

Transition type	$H - taken$	$C - active$	$H - enables$	$T - trigger$	$C - timers$	$MClock$
Initial State	{}	{}	$\{in1\}$	{}	{}	0
Configuration	$\{in1\}$	$\{S\}$	{}	{}	{1}	1
Configuration	$\{in1\}$	{}	{e1}	{3}	{}	2
Time	$\{in1\}$	{}	{e1}	{2}	{}	3
Time	$\{in1\}$	{}	{e1}	{1}	{}	4
Configuration	$\{in1, e1\}$	$\{a\}$	{}	{}	{3}	5
Time	$\{in1, e1\}$	$\{a\}$	{}	{}	{2}	6
Time	$\{in1, e1\}$	$\{a\}$	{}	{}	{1}	7
Configuration	$\{in1, e1\}$	{}	$\{e2\}$	{0}	{}	8
Configuration	$\{in1, e1, e2\}$	$\{E\}$	{}	{}	{1}	9
Configuration	$\{in1, e1, e2\}$	{}	{}	{}	{}	10

Table 7.2: Delayed Responsibility Execution

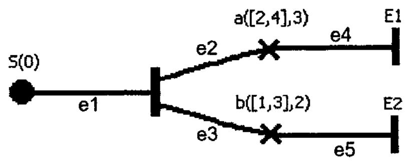


Figure 7.6: Parallel Flows

Transition type	$H - taken$	$C - active$	$H - enables$	$T - trigger$	$C - timers$	$MClock$
Initial State	{}	{}	$\{in1\}$	{}	{}	0
Configuration	$\{in1\}$	$\{S\}$	{}	{}	{1}	1
Configuration	$\{in1\}$	{}	{e1}	{0}	{}	2
Configuration	$\{in1, e1\}$	$\{OF\}$	{}	{}	{1}	3
Configuration	$\{in1, e1\}$	{}	$\{e2, e3\}$	{3, 2}	{}	4
Time	$\{in1, e1\}$	{}	$\{e2, e3\}$	{2, 1}	{}	5
Configuration	$\{in1, e1, e3\}$	$\{b\}$	$\{e2\}$	{1}	{2}	6
Time	$\{in1, e1, e3\}$	$\{b\}$	$\{e2\}$	{0}	{1}	7
Configuration	$\{in1, e1, e3, e2\}$	$\{a\}$	$\{e5\}$	{0}	{2}	8
Time	$\{in1, e1, e3, e2\}$	$\{a\}$	$\{e5\}$	{0}	{1}	9
Configuration	$\{in1, e1, e3, e2, e5\}$	$\{E2\}$	$\{e4\}$	{0}	{1}	10
Configuration	$\{in1, e1, e3, e2, e5, e4\}$	$\{E1\}$	{}	{}	{1}	11
Configuration	$\{in1, e1, e3, e2, e5, e4\}$	{}	{}	{}	{}	12

Table 7.3: Parallel Flow Execution

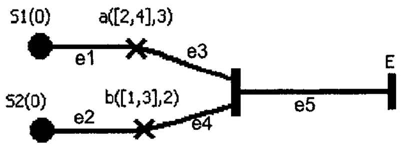


Figure 7.7: Synchronization of Flows

Transition type	$H - taken$	$C - active$	$H - enables$	$T - trigger$	$C - timers$	$MClock$
Initial State	$\{\}$	$\{\}$	$\{in1, in2\}$	$\{0,0\}$	$\{1\}$	0
Configuration	$\{in1\}$	$\{S1\}$	$\{in2\}$	$\{0\}$	$\{1\}$	1
Configuration	$\{in1, in2\}$	$\{S2\}$	$\{e1\}$	$\{3\}$	$\{1\}$	2
Configuration	$\{in1, in2\}$	$\{\}$	$\{e1, e2\}$	$\{2,2\}$	$\{\}$	3
Time	$\{in1, in2\}$	$\{\}$	$\{e1, e2\}$	$\{1,1\}$	$\{\}$	4
Configuration	$\{in1, in2, e1\}$	$\{a\}$	$\{e2\}$	$\{0\}$	$\{3\}$	5
Time	$\{in1, in2, e1\}$	$\{a\}$	$\{e2\}$	$\{0\}$	$\{2\}$	6
Time	$\{in1, in2, e1\}$	$\{a\}$	$\{e2\}$	$\{0\}$	$\{1\}$	7
Configuration	$\{in1, in2, e1, e2\}$	$\{b\}$	$\{e3\}$	$\{0\}$	$\{2\}$	8
Time	$\{in1, in2, e1, e2\}$	$\{b\}$	$\{e3\}$	$\{0\}$	$\{1\}$	9
Configuration	$\{in1, in2, e1, e2\}$	$\{\}$	$\{e3, e4\}$	$\{0,0\}$	$\{\}$	10
Configuration	$\{in1, in2, e1, e2, e3, e4\}$	$\{OF\}$	$\{\}$	$\{\}$	$\{1\}$	11
Configuration	$\{in1, in2, e1, e2, e3, e4\}$	$\{\}$	$\{e5\}$	$\{1\}$	$\{\}$	12
Configuration	$\{in1, in2, e1, e2, e3, e4, e5\}$	$\{E\}$	$\{\}$	$\{\}$	$\{1\}$	13
Configuration	$\{in1, in2, e1, e2, e3, e4, e5\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	14

Table 7.4: Run of Synchronized Flows

Three possible conditions can be distinguished:

- Condition 1: $(t \leq \delta)$. This condition triggers a configuration transition.

- Condition 2: $(c \leq \delta)$. This condition triggers a configuration transition.

- Condition 3: $(t > \delta) \wedge (c > \delta)$. This condition triggers a time transition.

Similarly to interleaving semantics, a special attention should be paid when an AND-Join is encountered.

Initial State: σ_{init} is defined with the following valuation: H-taken = $\emptyset$, C-active = $\emptyset$, H-enabled = incoming(StartPoints), T-trigger = delay(H-enabled), C-timers = $\emptyset$, MClock = 0.

All triggered edges are selected and taken in one single transition. In the given system state, the configuration transition (rule 4) is executed if all the following conditions hold:

(a) Condition 1 or Condition 2

(b) $\forall$ e $\in$ H-enabled, type(target(e))$\neq$AJ

$$\frac{\begin{array}{c}\textbf{Rule 4}\ \textit{h:=\{ all e} \in \textit{H-enabled, such that delay(e)}\leq\delta\} \\ \textit{H'-taken:=H-taken} \cup \{h\} \\ \textit{C'-active:=triggered(target(h))} \\ \textit{H'-enabled:=H-enabled} \cup \textit{enables(C-active)-}\{h\} \\ \textit{T'-trigger:= (T-trigger-}\delta\textit{)} \cup \textit{delay(target(H'-enabled))} \\ \textit{C'-timers:= duration(C'-active)} \\ \textit{MClock':=MClock} + \delta\end{array}}{\textit{(H-taken, C-active, H-enabled, C-timers, T-trigger,MClock)}\rightarrow\textit{(H'-taken, C'-active, H'-enabled, C'-timers', T'-trigger, MClock')}}$$

In given system state, the configuration transition (rule 5) is executed if all the following conditions hold:

(a) Condition 1 or Condition 2

(b) ∃ e ∈ H-enabled, such that type(target(e))=AJ and incoming(target(e))⊆ H-enabled

Rule 5 *h:={all e ∈ H-enabled, such that type(target(e))=AJ and incoming(target(e))⊆ H-enabled}*

$$\frac{\begin{array}{c} H'\text{-}taken{:=}H\text{-}taken \cup h \\ C'\text{-}active{:=}triggered(target(h)) \\ H'\text{-}enabled{:=}H\text{-}enabled \cup enables(C\text{-}active)\text{-}\{h\} \\ T'\text{-}trigger{:=}\ (T\text{-}trigger\text{-}\delta) \cup delay(target(H'\text{-}enabled)) \\ C'\text{-}timers{:=}\ duration(C'\text{-}active) \\ MClock'{:=}MClock + \delta \end{array}}{\begin{array}{c}(H\text{-}taken,\ C\text{-}active,\ H\text{-}enabled,\ C\text{-}timers,\ T\text{-}trigger,\ MClock) \rightarrow (H'\text{-}taken,\ C'\text{-}active,\ H'\text{-}enabled, \\ C'\text{-}timers,\ T'\text{-}trigger,\ MClock')\end{array}}$$

In given system state, the configuration transition (rule 4) is executed if all the following conditions hold:

(a) Condition 1 or Condition 2

(b) ∃ e ∈ H-enabled, such that type(target(e))=AJ and incoming(target(e))⊊ H-enabled

In given system state, the time transition (rule 6) is executed if all the following conditions hold:

(a) Condition 3 holds

(b) ∀ e ∈ H-enabled, type(target(e))≠AJ

Rule 6

$$\frac{\begin{array}{c} C'\text{-}timers{:=}C\text{-}timers\text{-}\delta \\ T'\text{-}trigger{:=}T\text{-}trigger\text{-}\delta \\ MClock'{:=}MClock + \delta \end{array}}{\begin{array}{c}(H\text{-}taken,\ C\text{-}active,\ H\text{-}enabled,\ C\text{-}timers,\ T\text{-}trigger,\ MClock) \rightarrow (H\text{-}taken,\ C\text{-}active,\ H\text{-}enabled, \\ C'\text{-}timers,\ T'\text{-}trigger,\ MClock')\end{array}}$$

Note: In time transitions, AND-Join is treated the same way as any other UCM construct. Note: Runs of the true concurrency semantics model have less states compared to the same runs in the interleaving semantics.

7.3.7 Example of CTS run

Figure 7.8 illustrates the CTS run of the UCM presented in Figure 7.6 according to true concurrency semantics.

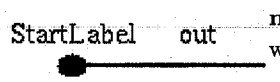
Transition type	$H - taken$	$C - active$	$H - enables$	$T - trigger$	$C - timers$	$MClock$
Initial State	$\{\}$	$\{S\}$	$\{\}$	$\{\}$	$\{1\}$	0
Configuration	$\{\}$	$\{\}$	$\{e1\}$	$\{0\}$	$\{\}$	1
Configuration	$\{e1\}$	$\{OF\}$	$\{\}$	$\{\}$	$\{1\}$	2
Configuration	$\{e1\}$	$\{\}$	$\{e2, e3\}$	$\{3, 2\}$	$\{\}$	3
Time	$\{e1\}$	$\{\}$	$\{e2, e3\}$	$\{2, 1\}$	$\{\}$	4
Configuration	$\{e1, e3\}$	$\{b\}$	$\{e2\}$	$\{1\}$	$\{2\}$	5
Configuration	$\{e1, e3, e2\}$	$\{b, a\}$	$\{\}$	$\{\}$	$\{1, 3\}$	6
Configuration	$\{e1, e3, e2\}$	$\{a\}$	$\{e5\}$	$\{0\}$	$\{2\}$	7
Configuration	$\{e1, e3, e2, e5\}$	$\{a, E2\}$	$\{e4\}$	$\{0\}$	$\{1, 0\}$	8
Configuration	$\{e1, e3, e2, e5, e4\}$	$\{E1\}$	$\{\}$	$\{\}$	$\{1\}$	9
Configuration	$\{e1, e3, e2, e5, e4\}$	$\{\}$	$\{\}$	$\{\}$	$\{\}$	10

Figure 7.8: Parallel Flow Execution in True Concurrency Mode

7.4 Formal Semantics of Timed UCM Models in Terms of ASM

When slightly modified, most of the untimed ASM rules presented in Section 4.2.1 can be applied to timed UCMs, except for start points, responsibilities and timers. Indeed, only a global clock advance statement (i.e. MClock:=MClock +1) is added to all ASM rules of UCM control constructs (OR-Fork, AND-Fork, etc.) to reflect the fact that these constructs take a single clock tick to complete. A start point, part of a plug-in map and bound to a stub entry point, should be executed without delay (similar to untimed start point) since stubs are used for structuring purpose. In the contrary, root map start points as well as unbound start points (part of a plug-in map) may be delayed. In what follows, we present the timed ASM rules for start points, responsibilities and timers.

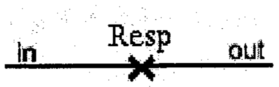
- **Start points (part of a root map or part of a plug-in map and bound to a stub entry point).** If the control is on the incoming edge (i.e. *in*), the *PreCondition-set* is satisfied, there occurs at least one event from the *triggeringEvent-set* and no additional delay is required (i.e. minDL $\leq$MClock $\leq$ maxDL), then the start point is triggered and the control passes to the outgoing edge. Figure 7.9 describes the start point rule.



```
if CurrConstruct is StartPoint(PreCondition-set, TriggerringEvent-set, StartLabel, in, out, minDL, maxDL) then
  if (EvaluatePreConditions & EvaluateTrigger & ( minDL ≤MClock ≤ maxDL)
then MClock := MClock + random(minDL,maxDL)
me.active:= out
where:
  - EvaluateTrigger: TriggerringEvent-set × {events} → Boolean; evaluates whether the set of events occurring at StartPoint are included in the TriggeringEvent-set.
  - EvaluatePreConditions: PreCondition-set → Boolean evaluates whether all preconditions are satisfied.
```

Figure 7.9: ASM Rule of Timed Start Point

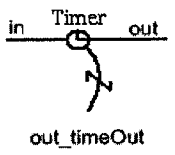
- **Responsibilities.** If the control is on the incoming edge (i.e. *in*) then the master clock *MClock* is increased by the value of the delay (i.e. random(minDL,maxDL)). After *Resp* is executed, *MClock* is increased by a random value from interval [*minDur*,*maxDur*] and the control passes to the outgoing edge.



```
if CurrConstruct is Responsibility (in, Resp, out, minDL, maxDL, minDur,
maxDur) then
    MClock:= MClock + random(minDL,maxDL)
    Resp
    me.active:= out
    MClock:= MClock+ random(minDur,maxDur)
```

Figure 7.10: ASM Rule of Timed Responsibility

- **Timer.** The timer rule is modified to reflect the increase of the global clock *MClock* (see Figure 7.11).



```
if CurrConstruct is Timer(in, TriggerringEvent-set, out, out_timeout, TO)
then
    if (Triggered) then me.active:= out
        MClock:= MClock+ random(0,TO)
    else me.active := out_timeout
        MClock:= MClock+TO
where Triggered: TriggerringEvent-set→Boolean determines whether a trig-
ger occurs within a predefined time frame.
```

Figure 7.11: ASM Rule of Timer

7.4.1 An AsmL Implementation of Timed UCM Semantics

Most of the data structures that have been presented in Section 4.3.2 are applicable to both untimed and timed UCM specification. Figure 7.12 shows the addition of time constraints to start point, responsibility and timer constructs.

structure UCMConstruct
   case SP_Construct
     in_hy **as** HyperEdge
     out_hy **as** HyperEdge
     label **as** String
     preCondition **as** BooleanExp
     minDL **as** Integer
     maxDL **as** Integer
     location **as** Component
 	
   case SP_PL_Construct
     in_hy **as** HyperEdge
     out_hy **as** HyperEdge
     label **as** String
     preCondition **as** BooleanExp
     location **as** Component
...
...
	case R_Construct
	   in_hy **as** HyperEdge
	   out_hy **as** HyperEdge
	   label **as** String
	   minDL **as** Integer
	   maxDL **as** Integer
	   minDur **as** Integer
	   maxDur **as** Integer
	   location **as** Component
	case Timer
	   in_hy **as** HyperEdge
	   Selec **as** Set of OR_Selection
	   label **as** String
	   TO **as** Integer
	   location **as** Component
	...
	...

Figure 7.12: Timed UCMConstruct Data Structure

Assuming a single-agent based solution with interleaving semantics, Figure 7.13 presents the timed version of ASM-UCM program (modifications are in underlined font). Contrary to the untimed version where the next edge to be executed is chosen randomly from the set of active edges, in the timed version the choice of subsequent edge is based on the delay of the target construct (next construct to be executed). Indeed, the edge leading to the construct with the minimum delay is selected.

7.4.2 Applying AsmL Semantics to the Simplified Telephone System (timed version)

Figure 7.14 shows a timed version of the trace presented in Figure 5.8. We assume that the root map start point has a delay within [2,5], responsibilities have a delay within [3,6] and a duration within [2,15].

```
class Agent
   const id as String
   var active as Edge
   var mode as Mode
Program()
step
  until ((act = {}) or (me.mode = inactive))
  do
   let h = {t1.edge || t1 in act }
   let del = {t1.delay || t1 in act }
   let minimumDL = (min x || x in del)
   choose z in act where z.delay= minimumDL
    choose h in level.ele where HyperExists(active, GetInEdge(h.source))
     match (s2.source)
// Rule of Start Point
     SP_Construct (a,b,c,d,e,f,g): step
         if d.Value() = true and (MClock ≤ f) and (MClock ≥ e)
          MClock := MClock + random(e,f)
          add activ(b, z.level, GetDelayTargetConstruct(b, z.level)) to act
          choose r in act where r.edge = a
          remove r from act
         else
          MClock := MClock + 1
// Rule of Plug-in Start Point
     SP_PL_Construct (a,b,c,d,e): step
         if d.Value() = true
          add activ(b, z.level, GetDelayTargetConstruct(b, z.level)) to act
          choose r in act where r.edge = a
          remove r from act
          MClock := MClock + 1
         else
          me.mode := inactive
// Rule of Responsibility
      R_Construct (a,b,c,d,e,f,g,l): step
          MClock := MClock + random(d,e)
         step
          ExecuteResponsibility((s2.source) as R_Construct)
         step
          MClock := MClock + random(f,g)
         add activ(b, z.level, GetDelayTargetConstruct(b, z.level)) to act
         choose r in act where r.edge = a
         remove r from act
// Rule of Timer
      TM_Construct (a,b,c,d,e): step
        choose v in b where (v.out_cond).Value() = true
         add activ(v.out_hy, z.level, GetDelayTargetConstruct(v.out_hy, z.level)) to act
          choose r in act where r.edge = a
          remove r from act
          MClock := MClock + random(0,d)
         ifnone
         choose v2 in b where (v2.out_cond).Value() = false
         MClock := MClock + d
         add activ(v2.out_hy, z.level, GetDelayTargetConstruct(v2.out_hy, z.level)) to act
         choose r in act where r.edge = a
         remove r from act
// ...
```

Figure 7.13: Timed ASM-UCM Program

```
This timed trace is generated with the following initial values:
subCND:True
subOCS:True
InOCSList:False
busy:False
```

```
Start Executing: Telephone System:Req
MClock=0
MClock=1
Start Point:Req in Component:UserOrig; MClock=2
Stub_Construct: SOrig; MClock=7
Plugin: Orig_plugin; MClock=8
Start Point:Start in Component:AgentOrig
Stub_Construct: Sscreen; MClock=9
Plugin: OCS_plugin; MClock=10
Start Point:Start in Component:AgentOrig;MClock=11
Start Executing Responsibility: checkOCS in component: AgentOrig at MClock=17
OR-Fork: OCS_OF1;MClock=32
End point: success in Component:AgentOrig; MClock=34
Start Executing Responsibility: snd_req in component: AgentOrig at MClock=40
End point: success in Component:AgentOrig
Stub_Construct: Sterm;MClock=56
Plugin: term_Plugin; MClock=57
Start Point:Start in Component:AgentTerm
OR-Fork: term_OF1;MClock=58
AND-Fork: term_AF1;MClock=59
Stub_Construct: Sdisplay;MClock=60
Plugin: CND_Plugin;MClock=61
Start Point:Start in Component:AgentTerm
AND-Fork: CND_AF1;MClock=62
End point: success in Component:AgentTerm;MClock=64
Start Executing Responsibility: display in component: AgentTerm at MClock=70
End point: disp in Component:AgentTerm;MClock=86
End point: display in Component:AgentTerm;MClock=87
End Point: display part of root map reached in Component:UserTerm; MClock=87
Start Executing Responsibility: ringTreatment in component: AgentTerm at MClock=93
End point: success in Component:AgentTerm;MClock=98
End Point: ring part of root map reached in Component:UserTerm; MClock=98
Start Executing Responsibility: ringingTreatment in component: AgentTerm at MClock=104
End point: reportSuccess in Component:AgentTerm;MClock=120
Start Executing Responsibility: fwd_sig in component: AgentOrig at MClock=126
End Point: ringing part of root map reached in Component:UserOrig;MClock=141
```

Figure 7.14: Timed Trace

7.5 Formal Semantics of Timed UCM Models in Terms of Timed Automata

Many approaches have been introduced to model timing behaviour of real-time systems, mostly derived from conventional finite state automata which are expanded to describe timing properties of the transition behavior. Although a number of different models have been proposed, e.g., several classes of *timed transition systems* [HMP91], *timed graphs* [ACD93], *event clock automata* [AFH94], *state clock automata* [RS97], *timed transition graphs* [SE94], *quantitative temporal structures* [FKG96], *interval structures* [RK97], or *statecharts* [Har87], the timed automata model of Alur and Dill [AD94] has become the standard.

7.5.1 Timed Automata

The theory of timed automata was introduced by Alur and Dill [AD94]. A timed automaton is a finite-state Büchi automaton extended with a finite set of real-valued variables modeling clocks. In its original version [AD90], constraints on the clock variables are used to restrict the behavior of an automaton, and Büchi accepting conditions are used to enforce progress properties. A simplified version, namely *Timed Safety Automata* is introduced in [HNSY94], to specify progress properties using local invariant conditions. *Timed Safety Automata* has been adopted in several verification tools for timed automata including UPPAAL [LPY97], SGM [WH02], RED [WWH05], KRONOS [Yov97] and HyTech [HHWT97].

A timed automaton is structured as a directed graph containing a finite set of nodes (called *locations*) and a finite set of labeled edges (called *transitions*). The logical clocks in the system are initialized with zero and then increase synchronously at the same rate. Clock constraints, represented by guards on edges, are used to restrict the behavior of the automaton. A guard is only an enabling condition of the transition and cannot force the transition to be taken. In the initial work by Alur and Dill [AD90], to avoid the fact that an automaton stays forever in any location, a subset of the locations in the automaton are marked as accepting (called Büchi-acceptance conditions) and only those executions passing through an accepting location infinitely often are considered valid behaviors of the automaton. Instead of accepting conditions, in timed safety automata [HNSY94], locations may be add local timing constraints called location invariants. An automaton may remain in a location as long as the clocks values satisfy the invariant condition of this location. Transitions may be labeled with an action and clocks may be reset to zero, when a transition is taken. Transitions occur instantaneously. Semantically, a state of a timed automaton is a recording of its present location and the readings of all clocks.

Figure 7.15 shows a timed automaton with two locations and two transitions. The timing behavior of the automaton is controlled by a clock *a*. The transition from the *Initial location* to *Location* won't occur unless a $>$ 2 invalidating the invariant a$\leq$2'. The same transition is guarded by a condition a$\geq$4 making the transition possible only if the local clock is greater or equal to 4. During the transition from *Location* to *Initial Location*, the clock *a* is reset.

In the following section, we give the formal syntax and semantics of timed automata as defined

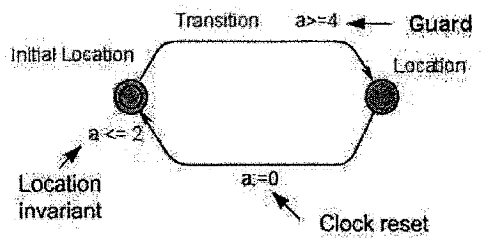


Figure 7.15: Example of a Timed Automaton

in [BY03].

7.5.2 Timed Automaton Formal Syntax and Semantics

Assume a finite set of real-valued variables C ranged over x, y, etc. that represent clocks and a finite alphabet Σ ranged over by a, b, etc. representing actions.

Clock constraints. A clock constraint is a conjunctive formula of atomic constraints of the form x $\sim$ n or x - y$\sim$n for x, y $\in$ C, $\sim \in \{\leq, <, =, >, \geq\}$ and n$\in\mathbb{N}$. A clock constraint is downward closed if $\sim \in \{\leq, <, =\}$. We use $\mathscr{B}(C)$ to denote the set of clock constraints, ranged over by g.

Guards and Invariants. A guard is a finite conjunction over data constraints and clock constraints. An invariant is a finite conjunction over downward closed clock constraints. Both contain additionally the constants true and false.

Assignments. A data assignment is of the form v:= A, where v$\in$V and A is an arithmetic expression over V. A clock reset is of the form x:=0, where x$\in$C.

Definition 10 (Timed Automaton) *A timed automaton A is a tuple* $\langle N, l_0, C, \Sigma, E, I\rangle$ *where:*

- *N is a finite set of locations (or nodes),*
- *$l_0 \in N$ is the initial location,*
- *C is the set of clocks,*
- *Σ is a set of actions,*
- *$E \subseteq N \times \mathscr{B}(C) \times \Sigma \times 2^C \times N$ is the set of edges connecting different locations*
- *I: $N \rightarrow \mathscr{B}(C)$ assigns invariants to locations*

There are two types of transitions between states: *delay transitions* (the automaton stays in a location) and *action transition* (an enabled edge is taken).

Clock assignments functions are used to track the changes of clock values. Let u, v denote such functions, and use $u \in g$ to mean that the clock values denoted by u satisfy the guard g. For d $\in \mathbb{R}_+$, let $u + d$ denote the clock assignment that maps all $x \in C$ to $u(x) + d$, and for $r \subseteq C$, let [r $\mapsto$ 0] u denote the clock assignment that maps all clocks in r to 0 and agree with u for the other clocks in C-$\{r\}$.

Definition 11 (Timed Automaton Operational Semantics.) *The semantics of a timed automaton is a transition system (also known as a timed transition system) where states are pairs $\langle l,u \rangle$ and transitions are defined by the rules:*

- *Delay transitions correspond to the elapsing of time while staying at some location: $\langle l,u \rangle \xrightarrow{d} \langle l,u+d \rangle$ if $u \in I(l)$ and $(u+d) \in I(l)$ for a non-negative real d in $\mathbb{R}_+$*

- *Action transition correspond to the execution of a transition from E: $\langle l,u \rangle \xrightarrow{a} \langle l',u' \rangle$ if $l \xrightarrow{g,a,r} l'$, $u \in g$, $u'=[r \mapsto 0]u$ and $u' \in I(l')$*

Definition 12 (Timed action and timed trace) *A timed action is a pair (t,a), where $a \in \Sigma$ is an action taken by an automaton $\mathscr{A}$ after $t \in \mathbb{R}_+$ time units since $\mathscr{A}$ has been started. The absolute time t is called* time-stamp *of the action a. A* timed trace *is a (possibly infinite) sequence of timed actions $\xi=(t_1,a_1)(t_2,a_2)\ldots(t_i,a_i)\ldots$ where $t_i \geq t_{i+1}$ for all $i \geq 1$.*

Definition 13 (Run of Timed Automaton) *A run of a timed automaton $\mathcal{A}= \langle N, l_0, C, \Sigma, E, I \rangle$ with initial state $\langle l_0,u_0 \rangle$ over a timed trace $\xi=(t_1,a_1)(t_2,a_2)(t_3,a_3)\ldots$ is a sequence of transitions:*
$\langle\, l_0,u_0 \,\rangle \xrightarrow{d_1} \xrightarrow{a_1} \langle\, l_1,u_1 \,\rangle \xrightarrow{d_2} \xrightarrow{a_2} \langle\, l_2,u_2 \,\rangle \xrightarrow{d_3} \xrightarrow{a_3} \langle\, l_3,u_3 \,\rangle \ldots$
satisfying the condition $t_i=t_{i-1} + d_i$ for all $i \geq 1$.

Definition 14 (Semantics of a network of Timed Automata) *Let $A_i= \langle\, N_i, l_i^k, C, \Sigma, E_i, I_i \rangle$ be a network of* n *timed automata. Let $\overline{l_0}=(l_1^0, \ldots, l_n^0)$ be the initial location vector. The semantics of A_i is defined as a transition system $\langle\, S, s_0, \rightarrow \,\rangle$ where $S = (N_1 \times \ldots \times N_n) \times \mathbb{R}_C$ is the set of nodes, $s_0=(\overline{l_0},u_0)$ is the initial state, and $\rightarrow \subset S \times S$ is the transition relation defined by:*

- *$(\bar{l},u) \rightarrow (\bar{l},u+d)$ if $\forall\, d'$: $0 \leq d' \leq d \Rightarrow u + d' \in I(\bar{l})$.*

- *$(\bar{l},u) \rightarrow (\bar{l}[l_i'/l_i],u')$ if there exists $l_i \xrightarrow{\tau g r} l_i'$ such that $u \in g$, $u'=[r \mapsto 0]u$ and $u' \in I(\bar{l})$.*

- *$(\bar{l},u) \rightarrow (\bar{l}[l_j'/l_j,l_i'/l_i],u')$ if there exists $l_i \xrightarrow{c?g_i r_i} l_i'$ and $l_j \xrightarrow{c?g_j r_j} l_j'$ such that $u \in (g_i \wedge g_j)$, $u'=[r_i \cup r_j \mapsto 0]u$ and $u' \in I(\bar{l})$.*

  such that $u \in g$, $u'=[r \mapsto 0]u$ and $u' \in I(\bar{l})$.

7.5.3 Timed Automaton Clock Region

Clock regions [ACD90, AD94], represent sets of clock assignments that are defined in order to obtain a finite representation for the infinite state space of a timed automaton.

Definition 15 (Extended State) *[AD94]. The behavior of a transition relation is determined by its state and the values of all its clocks. An extended state is a pair $\langle n, v \rangle$ where $n \in N$ and* v *is a clock interpretation for the set of clocks* C.

If two extended states, which correspond to the same location of the timed automata A, agree on the integral parts of all clocks values, and also on the ordering of the fractional parts of all clock values, then the runs starting from the two extended states are very similar. The integral parts of the clock values are needed to determine whether or not a particular clock constraint is met. whereas the ordering of the fractional parts is needed to decide which clock will change its integral part first. This is because clock constraints can involve only integers, and clocks increase at the same rate.

For example, let A be a timed automata with two clocks x and y. Let s be a location in A with an outgoing transition e to some other location. Consider two extended states ⟨n,v⟩ and ⟨n,v'⟩ in the transition relation that corresponds to the location n. Suppose that $v(x)=5.3$, $v(y)=7.5$, $v'(x)=5.5$ and $v'(y)=7.9$. Assume that the guard ϕ associated with e is $x\geq 8 \wedge y\geq 10$. It is easy to see if ⟨n, v⟩ eventually satisfies the guard, then so will ⟨n, v'⟩.

The integral parts of clock values can get arbitrarily large. But if a clock x is never compared with a constant greater than c, then its actual value, once it exceeds c, is of no consequence in deciding the allowed paths. For instance, if a clock x is never compared to a constant greater than 100 in the invariant associated with a location or in the guard of a transition. Then, it is impossible to distinguish between x having the value 101 or having the value 1001.

Alur and Dill [AD94] showed how to formalize this notion. For any $t \in \mathbb{R}$, *fract(t)* denotes the fractional part of t, and $\lfloor t \rfloor$ denotes the integral part of t; that is, $t = \lfloor t \rfloor + \text{fract}(t)$. We assume that every clock in C appears in some clock constraint. For each $x \in C$, let c_x be the largest integer c such that $(x \leq c)$ or $(c \leq x)$ is a subformula of some clock constraint appearing in E. The equivalence relation $\sim$ is defined over the set of all clock interpretations for C; $v \sim v'$ iff all the following conditions hold:

1. For all $x \in C$, either $\lfloor v(x) \rfloor$ and $\lfloor v'(x) \rfloor$ are the same, or both v(x) and v'(x) are greater than c_x
2. For all $x,y \in C$ with $v(x) \leq c_x$ and $v(y) \leq c_y$, $\text{fract}(v(x)) \leq \text{fract}(v(y))$ iff $\text{fract}(v'(x)) \leq \text{fract}(v'(y))$.
3. For all $x \in C$ with $v(x) \leq c_x$, $\text{fract}(v(x))=0$ iff $\text{fract}(v'(x))=0$

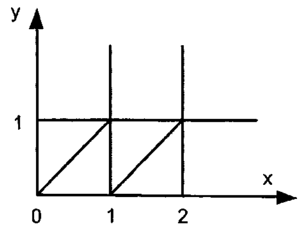


Figure 7.16: Clock Region Example

The equivalence classes of $\sim$ are called *regions* [AD94]. Each region can be represented by specifying:

1. For every clock x, one clock constraint from the set $\{ x=c \mid c = 0,1,\ldots c_x\} \cup \{ c-1 < x < c \mid c = 1,\ldots c_x\} \cup \{ x > c_x \}$,

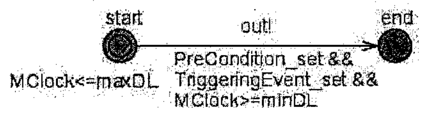
2. For every pair of clocks x and y such that c-1<x<c and d-1<y<d are clock constraints in the first condition for some c,d, whether *fract(x)* is less than, equal to, or greater than *fract(y)*.

Example: Figure 7.16 [AD94] shows the clock regions for a timed automaton with two clocks x and y where c_x=2 and c_y=1. In this example, there are a total of 28 regions: 6 corner points (e.g.[(1,0)]), 14 open line segments (e.g.[1<x<2 $\wedge$ y=x-1]), and 8 open regions (e.g.,[1<x<2 $\wedge$ 0<y<x-1]).

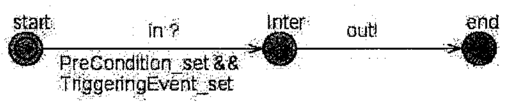
7.5.4 TA-based Semantics of Timed UCM Constructs

In this section, we model a timed UCM specification as a set of concurrent timed automata (called also processes). Processes interact with each other through synchronization channels and read-write operations to global variables. We intend to implement our proposed semantics using UPPAAL model checker [LPY97]. Since UPPAAL does not support *maximal progress semantics*, synchronization channels are used (instead of read-write operations to global variables) to coordinate the transfer of control between UCM constructs. A detailed discussion about UPPAAL and its semantics is given in Chapter 8. In what follows, we define a TA template for each timed UCM construct:

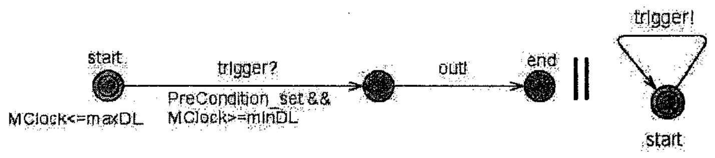
- **Start Point.** The start point is triggered when the *PreCondition-set* is satisfied, and there occurs at least one event from the *triggeringEvent-set*, and the delay constraint is met. This is described by a conjunction of Boolean conditions attached to the transition guard: PreCondition_set $\wedge$ triggeringEvent_set $\wedge$ (MClock$\geq$minDL). The location invariant *MClock* $\leq$ *maxDL* is used to make the process leave the start location whenever the master clock becomes greater than *maxDL*. The process writes into the channel *out* and the control passes to the next construct. Figure 7.17(a) illustrates the TA of a start point. If the start point is part of a plug-in map bound to a stub entry point, then the start point process must synchronize with the entry edge of the stub by reading from *in* channel, no time constraint is required (see Figure 7.17(b)).



(a) TA of Start Point



(b) TA of Plugin'Start Point

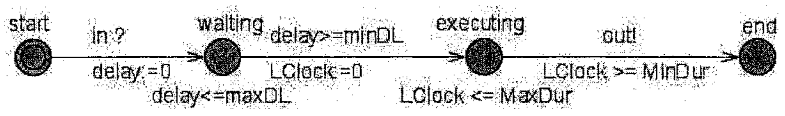


(c) Start point triggered by the environment

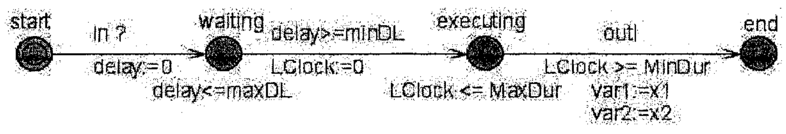
Figure 7.17: TA Templates for Start Points

Usually, a UCM describes the resulting interaction of a system and its environment in one single map (i.e., interactions between the different actors and the system under design). However, we may consider to model the environment in a separate map. For the sake of illustration, Figure 7.17(c) shows a start point that interacts with the environment through channel synchronization.

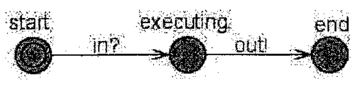
- **Responsibility.** Each responsibility has two local clocks:(1) *delay:* used to measure the delay that a responsibility may have and (2) *LClock:* used to monitor the execution duration of a responsibility. The process synchronizes with the preceding construct through the *in* channel. During this transition, local clock *delay* is reset (i.e., initialized to *zero*). The process stays in location *waiting* for an amount of time between *minDL* and *maxDL*, then start executing for an amount of time within [minDur, maxDur] interval. The location invariant $LClock \leq maxDur$ is used to make the process leave the *executing* state whenever the local clock becomes greater than *maxDur*. The control passes to the next construct after writing to the *out* channel (see Figure 7.18(a)). Responsibilities may have global variable assignments attached to them. These updates are attached to the transition between locations *executing* and *end.* Figure 7.18(b) illustrates such a TA with assignment 'var1:=x1' and 'var2:=x2'. Other variants of responsibilities may be considered (where the automaton is reduced to only three states and two transitions): Untimed responsibility with no clocks (Figure 7.18(c)), Atomic responsibility with a unique delay clock (Figure 7.18(d)) and Urgent responsibility with a unique LClock clock (Figure 7.18(e)).



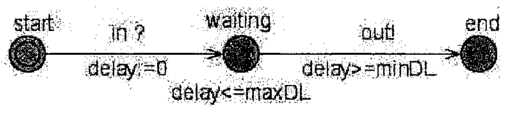
(a) TA of Responsibility



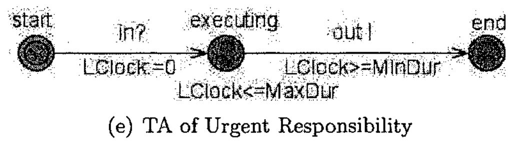
(b) TA of Responsibility with updates



(c) TA of untimed Responsibility



(d) TA of Atomic Responsibility



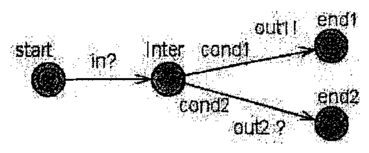
(e) TA of Urgent Responsibility

Figure 7.18: TA Templates for Responsibilities

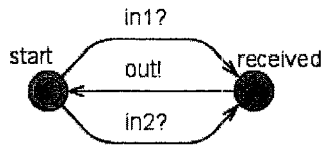
- **OR-Fork.** When the control passes to the OR-Fork through reading from *in* channel, the conditions are evaluated and the control passes to the edge associated with the true condition.

If more than one condition evaluates to true (i.e. nondeterministic choice), the control passes randomly to one of the outgoing edges associated to the true conditions. Figure 7.19(a) illustrates the automaton of an OR-Fork with two outgoing edges.

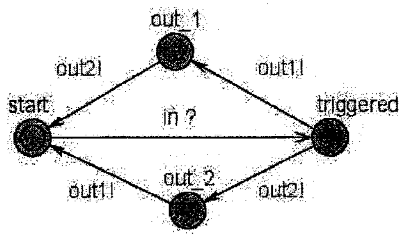
- **OR-Join.** When one or many flows reach an OR-Join (i.e., through synchronization on *in* channels), the control passes to the outgoing edge through *out* channel. Figure 7.19(b) shows the automaton of OR-Join with two incoming edges.



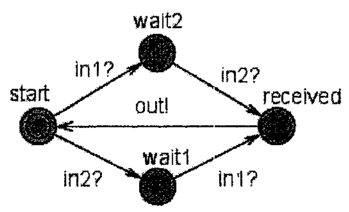
(a) TA of an OR-Fork



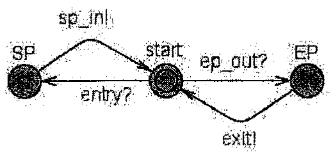
(b) TA of an OR-Join



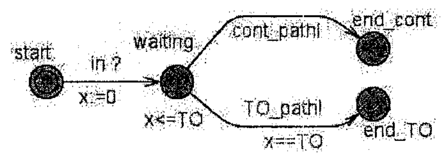
(c) TA of an AND-Fork



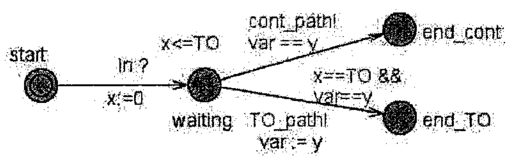
(d) TA of an AND-Join



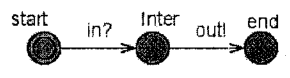
(e) TA of Stub



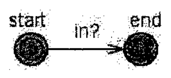
(f) TA of Timer



(g) TA of Timer with action



(h) TA of a plug-in end point



(i) TA of a root map end point

Figure 7.19: TA Templates of UCM Constructs

- **AND-Fork.** When the control reaches the AND-Fork (by reading from *in* channel), the process writes repeatedly to the outgoing channels. Figure 7.19(c) illustrates the automaton of an AND-Fork with two outgoing parallel flows.

- **AND-Join.** When parallel flows reach an AND-Join, it is required that the process reads from all incoming channels. The last flow arriving to the AND-Join will fire the outgoing transition (i.e. process writes into the *out* channel). Figure 7.19(d) shows the automaton of an AND-Join with two incoming parallel flows.

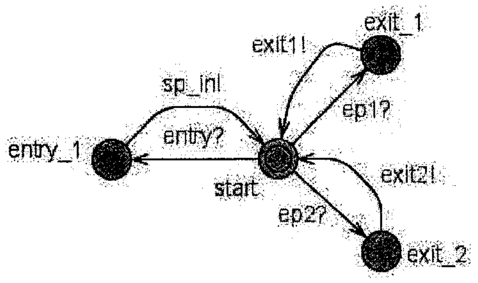
- **Stub.** The stub automaton implements the binding relation between a stub and a plug-in(i.e., B_s), allowing for the control to pass from a stub'entry point to a start point and from a plug-in end point to a stub'exit point. Figure 7.19(e) illustrates the timed automaton for a stub with one entry point *entry* and one exit point *exit.*

- **End Point.** If the end point is inside a plug-in, then the control passes to the stub's exit point bound to the plug-in end point (Figure 7.19(h)). Otherwise, the flow is stopped (Figure 7.19(i)).

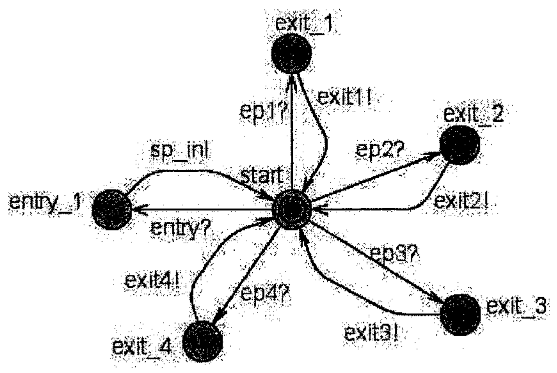
- **Timer.** The timer construct is illustrated in Figure 7.19(f). The timer stays for TO in location *waiting.* The control passes to the continuation path in case an event occurs before TO. Otherwise, the control moves to the time out path (i.e., TO_path). There are situations where an action is required as soon as the timer expires (i.e., timeout event and the action are atomic). Figure 7.19(g) shows a timer template with action (i.e. global variable assignment) attached to it.

7.5.5 Applying TA-based Semantics to the Simple Telephony System

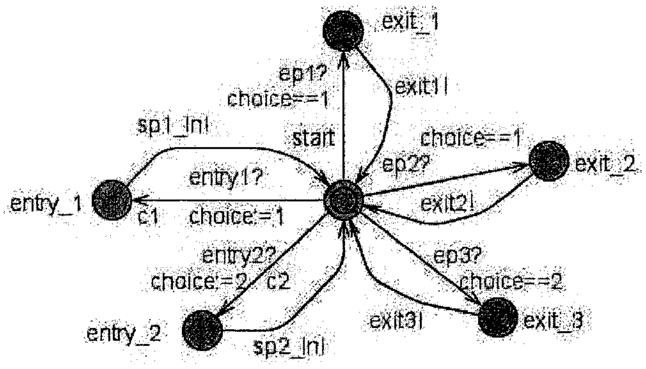
Our running example presents three types of stubs: static stub with one entry point and two exit points (i.e.,SOrig stub), static stub with one entry point and 4 exit points (i.e., Sterm Stub), dynamic stub with one entry point and two exit points which contains two plug-in maps (i.e., SScreen and SDisplay). Figure 7.20 shows the TA templates for these stub types.



(a) Static Stub with one entry and two exit point



(b) Static Stub with one entry and four exit points



(c) Dynamic Stub with one entry and two exit points (2 plug-in maps)

Figure 7.20: TA Templates for Stubs of the Simple Telephone System

Figure 7.21 shows the textual UPPAAL implementation of the simple telephone system introduced in Section 2.1.4. For an introduction to UPPAAL model checker [LPY97], the reader is referred to Section 8.1.5 in the subsequent chapter.

```
// Root map:
req = StartPoint(precond_SP, TriggEvent, e1, 1, 3);
Sorig = StaticStub1_2(e1, Orig_in1, Orig_success_out, e2, Orig_fail_out, e4);
notify = EndPoint(e4);
fwd_sig_busy = Responsibility(e5, e6, 1, 2, 1, 4);
fwd_sig_ringing = Responsibility(e7, e8, 1, 2, 1, 4);
busy_root = EndPoint(e6);
ringing_root = EndPoint(e8);
ring_root = EndPoint(e3);
display_root = EndPoint(e9);
// Originating plug-in:
Orig_start = StartPoint_plugin(Orig_in1, precond_SP, TriggEvent, O1);
SScreen = DynamicStub1_2_2plugins(O1, sub_OCS, OCS_in1, O1, not_sub_OCS, DEF_in1,
OCS_success_out, O2, OCS_fail_out,O4, DEF_continue_out, O2, ch1);
snd_req = Responsibility(O2, O3, 1, 2, 1, 4);
Orig_success = EndPoint_plugin(O3, Orig_success_out);
Orig_fail = EndPoint_plugin(O4, Orig_fail_out);
// OCS plug-in:
OCS_start = StartPoint_plugin(OCS_in1, precond_SP,TriggEvent, OCS1);
CheckOCS = Responsibility(OCS1,OCS2, 1, 2, 1, 4);
OCS_OF1 = OR_Fork(OCS2, InOCSList , OCS4, not_InOCSList, OCS3);
deny = Responsibility(OCS4,OCS5, 1, 2, 1, 4);
OCS_success = EndPoint_plugin(OCS3,OCS_success_out);
OCS_fail = EndPoint_plugin(OCS5,OCS_fail_out);
// Terminating plug-in:
Sterm = StaticStub1_4(e2, term_in1, Sterm_success_out, e3, Sterm_display_out, e9,
Sterm_busy_out, e5, Sterm_ringing_out, e7);
Sterm_start = StartPoint_plugin(term_in1, precond_SP,TriggEvent, T1);
term_OF1 = OR_Fork(T1, busy , T3, not_busy, T2);
term_AF1 = AND_Fork (T2, T5, T9);
SDisplay = DynamicStub1_2_2plugins(T5, sub_CND, CND_in1, T5, not_sub_CND, DEF_in1,
CND_success_out, T6, CND_disp_out, T8, DEF_continue_out, T6, ch2);
term_ringingTreatment = Responsibility(T9, T10, 1, 2, 1, 4);
term_reportSuccess = EndPoint_plugin(T10, Sterm_ringing_out);
term_busyTreatment = Responsibility(T3, T4, 1, 2, 1, 4);
term_ringTreatment = Responsibility(T6, T7, 1, 2, 1, 4);
term_fail = EndPoint_plugin(T4, Sterm_busy_out);
term_disp = EndPoint_plugin(T8, Sterm_display_out);
term_success = EndPoint_plugin(T7, Sterm_success_out);
// CND plug-in:
CND_start = StartPoint_plugin(CND_in1, precond_SP,TriggEvent, CND1);
CND_AF1 = AND_Fork (CND1, CND2, CND4);
CND_display = Responsibility(CND2,CND3, 1, 2, 1, 4);
CND_success = EndPoint_plugin(CND4,CND_success_out);
CND_disp = EndPoint_plugin(CND3,CND_disp_out);
// default plug-in:
DEF_start = StartPoint_plugin(DEF_in1, precond_SP,TriggEvent, DEF1);
DEF_continue = EndPoint_plugin(DEF1,DEF_continue_out);
// List one or more processes to be composed into a system.
system req, Sorig, Orig_start, SScreen, OCS_start, CheckOCS, OCS_OF1,
deny, OCS_success, snd_req, Orig_success, Sterm, Sterm_start, term_OF1, term_AF1,
term_ringingTreatment, term_reportSuccess, OCS_fail, Orig_fail, notify,
SDisplay, CND_start, CND_AF1, CND_display, CND_success, CND_disp,
fwd_sig_busy, fwd_sig_ringing, busy_root, term_ringTreatment, term_busyTreatment, term_fail,
DEF_start, DEF_continue, term_disp , term_success, ringing_root, ring_root, display_root;
```

Figure 7.21: UPPAAL Textual Implementation of the Simple Telephone System

7.5.6 Limitations

A timed UCM specification is represented as a collection of timed automata where each timed UCM construct is translated into an instance process based on the underlined templates. This design solution is simple to implement and provides a great level of flexibility. However, the following shortcomings are worth noting:

1. This approach is costly in terms of number of concurrent processes, number of locations, number of transitions and number of local clocks.

2. This approach does not support cycles (i.e., loops). Indeed, once a construct is executed (i.e., reaches its TA end location), it cannot be executed a second time because there is no extra transition connecting its *end* location to its *start* location. This limitation can be partially solved, if we merge the start and end locations of some UCM constructs. Figure 7.22 illustrates a responsibility that can be executed multiple times within a loop.

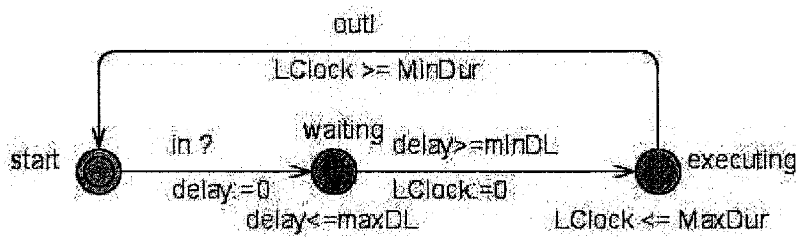


Figure 7.22: Responsibility that Supports Loops

In Chapter 8, we propose an approach that reduces considerably the number of processes and allows the description of cycles. The resulting optimized approach is used to verify properties using model checking.

7.6 Chapter Summary

In this chapter, we have extended the Use Case Maps language with time. We have introduced an approach to describe timing constraints in UCM based on the criteria discussed in Section 6.1. Three formalization approaches for timed UCM language were presented:

- CTS [MP96] based semantics: Based on a discrete time model, CTS provides an easy, natural and flexible way to reason about system execution over time. Indeed, the proposed transition rules provide an insight into the system state at every single clock tick which eliminates hidden ambiguities. Furthermore, we have defined two step semantics (i.e. two sets of transition rules) for timed UCM models, to cover both interleaving and true concurrency models. However, the major drawback of this approach is that there is a lack of tool support for CTS.

- ASM [Gur88] based semantics: Based on a discrete time model, we have extended the untimed ASM semantics introduced in Chapter 4 to cover time extensions. This approach has two advantages. First, it is relatively cheap to implement since it is built upon the untimed ASM operational semantics presented in Chapter 4). Second, it provides an environment (AsmL based) to simulate (one shot or step-by-step simulation) and to capture various aspects of a system run (e.g. execution time, executed constructs, components, etc.) in one single timed trace. However, this approach does not support true concurrency model semantics.

- TA [AD94] based semantics: Based on a dense time model, we have defined a timed automaton template for each timed UCM construct. Timed Automata (TA) formalism has proved to be rich enough to express timing constraints for many real-life examples. Furthermore, UCM models expressed in TA can be validated and verified using UPPAAL model checker [LPY97].

Extending UCMs with time represents a first step towards the construction of a formal framework for using UCM to describe, simulate, analyze and verify real-time systems at high level of abstraction. In the next chapter, we propose an approach to formally verify timed UCM specifications using model checking.

Chapter 8

Model Checking Timed UCM Specifications

In the previous chapter, we have extended Use Case Maps language with real-time constraints to allow for supporting quantitative analysis at early phases of the software development process. Model checking [EMCGP99] has proven to be a successful technology to verify requirements and design for a variety of real-time embedded and safety-critical systems. In this chapter[1], we combine model checking technique with UCM requirement slicing (introduced in Chapter 5) to formally verify timed Use Case Maps specifications.

The structure of this chapter is as follows: the subsequent section provides a brief introduction to model checking, temporal logics formalisms (standard temporal logic formalisms such as CTL*/CTL/LTL and real-time temporal logic such as MTL and TCTL) and supporting tools. Section 8.1.3 reviews the problem of state space explosion and provides an overview of existing techniques to cope with it. A full Section (Section 8.1.5) is devoted to UPPAAL model checker [LPY97] which is our selected verification tool. Section 8.2 describes our early stages verification approach, discusses sequential vs. parallel control flows and proposes a mechanism to sequentially compose timed automata in the context of UCM. A discussion on UPPAAL lack of maximal progress and how our proposed solution overcomes this limitation can be found in Section 8.4. Finally, we apply our proposed approach to our running case study of the simple telephone system (Section 8.5).

8.1 Model Checking

Model checking is a formal-verification technique based on state exploration. Given a state transition system and a property, model checking algorithms exhaustively explore the state space to determine whether the system satisfies the property. The essential idea behind model checking is shown in Figure 8.1. A model checker accepts a model of the specification and a property (called also specification) that the final system is expected to satisfy. The result is either a claim that the

[1]This chapter content is published in SDL Forum - SDL 2007) [HRD07a]

property is true or else a counterexample (a sequence of states from some initial state) falsifying the property. In practice, counter examples often provide valuable debugging information, and can be used by the software engineer to modify the specification, the model, or the property checked. The idea is that by ensuring that the model satisfies enough system properties, we increase our confidence in the correctness of the model.

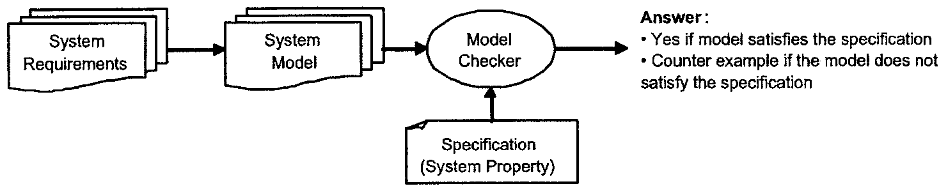

System Requirements
System Model
Model Checker
Specification (System Property)
Answer:
• Yes if model satisfies the specification
• Counter example if the model does not satisfy the specification

Figure 8.1: The Model Checking Approach

Model-checking operates on Kripke structures [Kri63], that is finite state automata with an additional labeling function associating atomic propositions with states.

Definition 16 (Kripke Structure) *A Kripke structure M is a 4-tuple $M = (S, S_0, R, L)$, where*

- *S is a finite set of states,*
- *$S_0 \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is a total transition relation, and*
- *L: $S \rightarrow 2^{AP}$ is a labeling function that labels each state with the set of atomic propositions (AP) true in that state.*

The formal definition of the model-checking problem is [EMCGP99]:

Definition 17 (The Model-Checking Problem) *Given a Kripke structure $M = (S, S_0, R, L)$, that represents a finite-state concurrent system and a temporal-logic formula f expressing some desired specification, find the set of all states in S that satisfy f: $\{ s \in S \sim M,s \models \mathrm{f} \}$ The system satisfies the specification provided that all the initial states are in the set.*

The model checking approach depends on the logic used for the specification. Each approach requires its own algorithm. Within the branching time logic, properties are related to sets of states. Thus, fixpoint computations can be used as efficient algorithms for a state space exploration in finite transition systems. Formulas of a linear temporal logic are related to single paths. Hence, the notion of semantical tableau [LP85] is adapted for linear temporal logics. In the next section, syntax and semantics of the most commonly used temporal logics (i.e. CTL, LTL) are introduced.

8.1.1 Temporal Logics

Temporal logic is used in order to specify properties of state transition systems (or Kripke structures). The logic uses atomic propositions and boolean operators such as conjunction, disjunction and negation to construct expressions that describe sequences of transitions between states. Temporal logic does not consider explicit time but rather a notion of sequences of states which describe possible computations (or behavior) of the system. Temporal logic differ in the semantics and the operators that they provide.

The Computational Tree Logic (CTL)

CTL [CES86] is a propositional branching-time temporal logic. Two types of formulas can be distinguished: (1) state formulas, expressing a property of a specific state, and (2) path formulas, modeling a proposition over a specific path. The set V denotes the set of atomic propositions (i.e., boolean variables for characterizing states).

The syntax of a CTL formula is given by the following rules.

Definition 18 (Syntax of CTL) .
State formulas:

i. If $\varphi \in V$, then φ is a state formula.

ii. If φ and ψ are state formulas, then $\neg\ \varphi$ and $\varphi \vee \psi$ are state formulas.

iii. If φ is a path formula, then $\boldsymbol{E}\varphi$ is a state formula.

Path formulas:

i. If φ and ψ are state formulas, then $\boldsymbol{X}\varphi$ and $\varphi\ \boldsymbol{U}\ \psi$ are path formulas.

ii. If φ is a path formula, $\neg\varphi$is a path formula.

Formulas are composed of *path quantifiers* and *temporal operators*. The path quantifier are used to describe the branching structure in the computation tree. There are two such quantifiers:

- **A**: For all computation paths
- **E**: For some computation path

These quantifiers are used in particular state to specify that all of the paths or some of the paths starting at that state have some property. There are four basic operators:

- **X** (*next time*): requires that a property holds in the second state of the computation path,
- **F** (*in the future*): asserts that a property will hold in some future step on a computation path,
- **G** (*globally*): specifies that a property holds at every state on the computation path,
- **U** (*until*): It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.

In CTL a state formula is not also a path formula (in contrast to CTL* which comprises CTL and LTL, see Section 8.1.1). Thus, temporal operators cannot be combined arbitrarily: **E** and **A** are only allowed for prefixing path formulas, whereas **X**, **U**, **F** and **G** are only allowed in combination with state formulas. For instance, formulas of the form $(\mathbf{X}\varphi)$ or $(\mathbf{A}(\mathbf{X}\varphi \vee \mathbf{F}\psi))$ are not allowed. Hence, eight basic temporal operators can be used in CTL: **AX**, **EX**, **AG**, **EG**, **AF**, **EF**, **AU**, and **EU**. These can be expressible with **EX**, **EU**, and **EG** only:

$$\begin{aligned}
\mathbf{EF}\,\varphi &\iff \mathbf{E}(\text{true}\ \mathbf{U}\ \varphi)\\
\mathbf{AX}\,\varphi &\iff \neg\mathbf{EX}(\neg\,\varphi)\\
\mathbf{AG}\,\varphi &\iff \neg\,\mathbf{EF}(\neg\,\varphi) \iff \neg(\mathbf{E}(\text{true}\ \mathbf{U}\ \neg\varphi))\\
\mathbf{AF}\,\varphi &\iff \neg\mathbf{EG}(\neg\,\varphi)\\
\mathbf{A}(\varphi\ \mathbf{U}\ \psi) &\iff \neg\,\mathbf{E}(\neg\,\psi\ \mathbf{U}\ \neg\,\varphi \wedge \psi) \wedge \neg(\mathbf{EG}(\neg\psi))
\end{aligned}$$

The four operators that are used widely are illustrated in Figure 8.2.

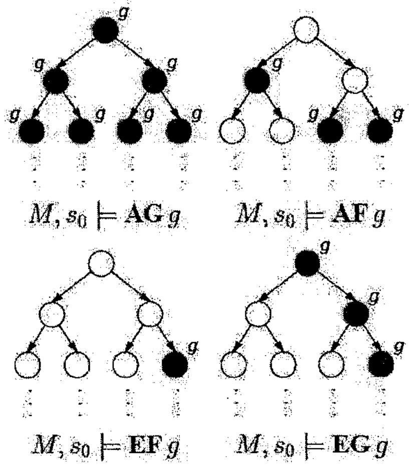

g
$M, s_0 \models \mathbf{AG}\,g$
$M, s_0 \models \mathbf{AF}\,g$
$M, s_0 \models \mathbf{EF}\,g$
$M, s_0 \models \mathbf{EG}\,g$

Figure 8.2: Basic CTL Operators

Some typical CTL formulas that might arise in verifying a finite state concurrent system are

- **Liveness:** All computation paths satisfy that if a request *req* occurs, then it will be eventually acknowledged (*ack*): **AG** (*req* $\rightarrow$ **AF** *ack*)
- **Safety:** In each state, which can be reached, system crash never occur (the bad thing will never occur). **AG**($\neg$crash)
- **Absence of Livelock:** In every state there is a path on which eventually the system is able to proceed. For instance, proceeding may be specified as being able to get to restart state, i.e., restart is satisfied. **AG**(**EF** restart)

Linear Temporal Logic (LTL)

Linear-temporal logic (LTL) [MP92] is related to linear rather than branching time. LTL consists of formulas that have the form $\mathbf{A}f$ where $\mathbf{f}$ is a path formula in which the only state subformulas permitted are atomic propositions.

The syntax of a CTL formula is given by the following rules.

Definition 19 (Syntax of CTL) . *With V as the set of atomic propositions, the syntax of an LTL formula is given by the following rules:*

i. If $\varphi \in V$, then φ is a path formula.

ii. If φ and ψ are path formula, then $\neg\, \varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\boldsymbol{X}\varphi$, $\boldsymbol{F}\varphi$, $\boldsymbol{G}\varphi$, $\varphi\ \boldsymbol{U}\psi$ are path formulas.

The Computational Tree Logic CTL*

CTL and LTL are two sublogics of CTL*. CTL* provides two different kinds of temporal operators. Linear temporal operators stating propositions with respect to a path and temporal operators of branching-time referring to several branching paths that start in the current state. These operators can be found in LTL and CTL sub-logics as well.

The syntax of a CTL* formulas is given by the following rules.

Definition 20 (Syntax of CTL*) .
State formulas:

i. If $\varphi \in V$, then φ is a state formula.

ii. If φ and ψ are state formulas, then $\neg\, \varphi$, $\varphi \vee \psi$ and $\varphi \wedge \psi$ are state formulas.

iii. If φ is a path formula, then $\boldsymbol{E}\varphi$ and $\boldsymbol{A}\varphi$ are state formula.

Path formulas:

i. If φ is a path formula, $\neg\varphi$is a path formula.

ii. If φ and ψ are state formulas, then $\not\psi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\boldsymbol{X}\varphi$, $\boldsymbol{F}\varphi$, $\boldsymbol{G}\varphi$, $\varphi\ \boldsymbol{U}\,\psi$ are path formulas.

The major difference to CTL is that in CTL* every state formula is a path formula as well. Thus, in CTL* every formula can be combined with the quantifying operators over paths (**E** and **A**) which does not hold for CTL. For example, the formula $\mathbf{E}\ (\varphi \wedge \mathbf{X}\psi)$ is syntactically correct in CTL* but not in CTL. In CTL every state formula has to be preceded by a path quantifier.

The three logics (CTL, LTL and CTL*) have different expressive powers [CD89, EH86]. For instance, There is no CTL formula that is equivalent to the LTL formula $\mathbf{A}(\mathbf{FG}\mathrm{p})$. This formula expresses the property that along every path, there is some state from which p will hold forever. Likewise, there is no LTL formula that is equivalent to the CTL formula $\mathbf{AG}(\mathbf{EF}\mathrm{p})$. The disjunction of the two formulas $\mathbf{A}(\mathbf{FG}\mathrm{p})\vee\ \mathbf{AG}(\mathbf{EF}\mathrm{p})$ is a CTL* formula that is not expressible in either CTL or LTL [EMCGP99]. Figure 8.3 depicts the expressiveness of the three temporal logics.

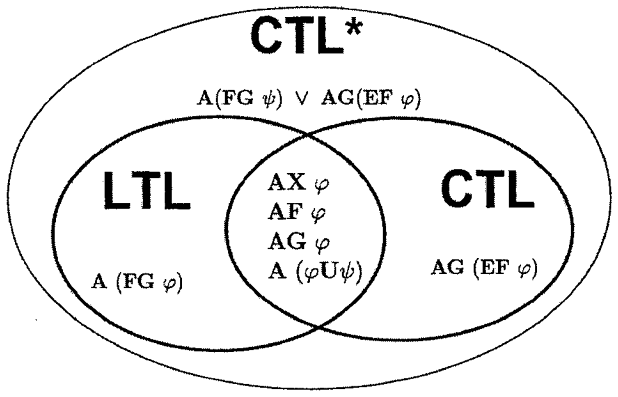


Figure 8.3: Expressiveness of the Three Temporal Logics

8.1.2 Real-time Temporal Logic

Standard temporal logics, such as CTL [CES86], ACTL [NV90] and LTL [MP92], which are subset of μ-calculus, are inadequate for real-time applications because they only deal with qualitative timing properties. Real-time temporal logics extend standard temporal logics with temporal operators that allow the definition of quantitative temporal relationships such as distance among events in time units.

In [AH90, BMN00] many real-time temporal logics have been surveyed and a series of criteria for assessing their capabilities was presented. Among these criteria are the logic expressiveness, the logic's order, decidability of the logic, the use of temporal operators, the fundamental time entity and the structure of time. In the following, we give a brief overview of MTL [Koy90] and TCTL [Alu92]. For a detailed description, we refer the reader to [Alu92, Koy90].

Metric Temporal Logic (MTL)

Metric temporal logic (MTL) [Koy90] is an extension to LTL [MP92] in which the temporal operators(always ($\Box$), eventually ($\diamond$), next ($\circ$), strong until ($\mathcal{U}$) and weak until($\mathcal{W}$)) are replaced by time-constrained versions. For example, the formula $\Box_{[0,k]}\varphi$ expresses that φ holds for the next k time units. MTL is interpreted over a discrete time line and assumes integer time. MTL is undecidable [AH90].

Timed Computational Tree Logic (TCLTL)

Timed computational tree logic (TCTL) proposed by Rajeev Alur [Alu92] is a propositional branching-time logic. TCTL extends computational tree logic [CES86] by allowing timing constraints on the temporal operators (always (G), eventually (F), strong until (U), and weak until (W) operators, which are either existentially (E) or universally (A) quantified). For example, the formula $AG(P \Rightarrow AF_k(S))$ expresses the time-bounded response property *'Globally, S responds to P within k time units'*. The semantics of TCTL is defined over a dense time line. MTL is decidable.

8.1.3 State Space Explosion

The main disadvantage of model checking is related the state explosion problem that can occur if the system being verified has many components that can make transitions in parallel. In this case the number of global system states may grow exponentially with the number of processes and it becomes computationally infeasible to store all states in memory or to produce a solution in a reasonable time. To solve the state explosion problem many approaches have been proposed:

- **Symbolic representation**: McMillan [McM92] proposed the use of symbolic representation for the state transition graphs to reduce the space of states to a smaller sub-set. So much larger systems could be verified. The new symbolic representation was based on Bryant's ordered binary decision diagrams (OBDDs) [Bry86]. In order to check the satisfiability of a given formula, one needs to consider true and false assignments to all the variables, and evaluate the whole expression for each such combination of all the variables. Such an approach results in computing a binary tree that has an exponential size in terms of the number variables. Bryant suggested reducing the tree to only those branches, which may have an effect on the outcome of the Boolean formula. For example, if the left side of an "or" operation has evaluated to one, then there is no need to evaluate the right side. The model checking system that McMillan developed is called SMV (Symbolic Model Verifier) [Bry86]. It is based on a language for describing hierarchical finite-state concurrent systems. The model checker extracts a transition system represented as an OBDD from a program in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies its properties.

- **Partial order reduction:** This technique exploits the independence of concurrently executed events [GP93, Val91]. Two events are independent of each other when executing them in either order results in the same global state.

- **Abstraction** [CGL92]: This technique is essential for reasoning about reactive systems that involve data paths. The abstraction is usually specified by giving a mapping between the actual data values in the system and a small set of abstract data values.

- **Symmetry** [CFJ93]: Finite state concurrent systems frequently contain replicated components, (such as protocols, a protocol may involve a network of identical communicating processes). Having symmetry in a system implies the existence of a non trivial permutation group that preserves the state transition graph. Such a group can be used to define an equivalence relation on the state space of the system and to reduce the state space.

- **Parallelization (or distribution) of the state space search**: To overcome the space problem of BDD-based model checkers, a promising approach is to parallelize (or distribute) the state space search to exploit the accumulative computation power and memory of a number of machines that work in parallel.Heyman et al. [HGGS00] proposed an based on an initial partitioning of the state space among all processes in the network and on a continuous load balancing that keeps the workload among the processes relatively balanced. Each process iteratively applies image computation to its set of new states N, exchanging non-owned states with other processes, and collecting o owned states in its set of reachable states R. Load balance is available at the end of each iteration. It balances the sizes of the sets of reachable states in the different processes. Later, Grumberg et al. [GHS06] have claimed that the success of Heyman et al. [HGGS00] approach strongly depends on an effective slicing procedure (Slicing is said to be effective if it avoids duplication and if it results in evenly split, smaller BDDs). To overcome this limitation, Grumberg et al. have proposed an algorithm that dynamically allocates and reallocates processes to tasks. Furthermore, the algorithm provide a recovery mechanism from local state explosion. The authors have claimed that, in addition to its efficiency, its high adaptability makes it suitable for exploiting the resources of very large and heterogeneous distributed, non-dedicated environments [GHS06].

- **Modular decomposition [SG91]**: The properties of a complex system are decomposed into properties that describe the behavior of small parts of the system. Local properties are checked using only the part of the system that it describes. If the conjunction of the local properties implies the overall specification, then the complete system must satisfy the system properties. This compositional reasoning is not feasible, when there are mutual dependencies between components. In such cases, when verifying a property of one component, assumptions should be made about the behavior of other components. The assumptions must later be discharged when the correctness of the other components is established.

- **Slicing [Wei84]**: Slicing, a program reduction technique, can be a useful way of reducing program size to allow more efficient model checking. The BANDERA tool [CDH$^+$00] is one of the examples of the use of slicing to reduce Java programs for model checking. Slicing alleviates the state explosion by removing system parts (based on the slicing criterion) that cannot affect the truth (or falsity) of the temporal logic formula. This is similar to the motivation for removing clauses in traditional logic programs: it reduces the length of computations of individual intentions.

8.1.4 Model Checking Tools

There are a wide variety of model checkers available, with a number of different capabilities suited for different kinds of problems. In what follows, we provide a brief overview of the most popular model checking tools. However, we devote a full section (Section 8.1.5 to UPPAAL [LPY97], which is the selected model checker in this thesis.

- **Spin [Hol97]:** SPIN was developed at Bell Labs starting in 1980. It is written in ANSI Standard C, and is portable across multiple platforms. Spin targets software verification, not hardware verification. In SPIN, the system models are described in a modeling language called PROMELA (Process Meta Language) where systems can be seen as a set of synchronized extended finite state machines. Spin works on-the-fly, which means that it avoids the need to preconstruct a global state graph, or Kripke structure, as a prerequisite for the verification of system properties. It offers a full LTL model checking system. To optimize the verification runs, the tool uses partial order reduction techniques, and (optionally) BDD-like storage techniques.

SPIN does not support quantitative timing relations. However, SPIN is still well suited for modeling the untimed aspects of the protocol processes and for expressing the relevant (untimed) properties. Tools like UPPAAL [LPY97] and KRONOS [Yov97] are more suitable for dealing with quantitative time constraints.

- **NuSMV [CCGR99]:** NuSMV, an open source software, has been developed as a joint project between ITC-IRST (Istituto Trentino di Cultura, Istituto per la Ricerca Scientifica e Tecnologica in Trento, Italy), Carnegie Mellon University, the University of Genoa and the University of Trento. NuSMV is an extension of the SMV symbolic model checker [McM92]. In NuSMV, the system models are described in SMV language, a simple circuit description language, allowing for reachability analysis, BDD based CTL model checking (under fairness), computation of quantitative characteristics of the model, and generation of counterexamples. Furthermore, NUSMV supports bounded model checking with LTL (including past operators). Other NuSMV features include deadlock checking, computing the number of reachable states and simulation. NuSMV is mainly used to verify digital circuits (or systems easily modeled as circuits).

- **Murφ**: Murφ [DDHY92] is an explicit model checker (represents states *explicitly* where each visited state is stored in the hash table). When a state is generated that is already in the hash table, the search algorithm does not expand its successor states (they were expanded whenever the state was originally inserted in the table). Murφ description language consists of declaration of constants, types, global variables and procedures; a collection of transition rules; a description of the initial states; and a set of invariants. Each transition rule is a guarded command which consists of a Boolean condition and an action that are both written in a Pascal-like language. A Murφ state is an assignment of values to all global variables of the description.

  To reduce the number of reachable states, Murφ uses symmetry reduction, reversible rules [ID96a] and repetition constructors [ID96b].

- **KRONOS [Yov97]:** KRONOS is a timed model checker developed by Sergio Yovine at VERIMAG. In KRONOS, systems are modeled by timed automata and properties can be specified either in TCTL-formula as a logical approach or in timed automata as a behavior approach. KRONOS implements a symbolic model-checking algorithm, where sets of states are symbolically represented by linear constraints over the clocks of the timed automaton.

  KRONOS provides both backward and forward analysis. To reduce the size of the explored state space, KRONOS optimizes the number of clocks in the model, uses on-the-fly exploration, partial-order techniques and binary decision diagrams. Since KRONOS is purely based on timed automaton, it does not support some data types such as boolean and integer variables. This limitation is addressed by UPPAAL [LPY97] which extends the timed automata with more general data types.

Other famous model checking tools include:

- Petri net tools: INA, Lola, PEP, Design/CPN, etc.

- Caesar Aldebaran (CADP): A set of model checking tools based on LTSs.

- Java Pathfinder 2: Model checker for Java programs.

- Bandera: Java abstraction and slicing system, with model checking back-ends.

- Slam: A (Microsoft) tool for model checking C programs.

8.1.5 The Model Checker UPPAAL

UPPAAL [LPY97] is an integrated tool environment for modeling, validation and verification of real-time systems modeled by a network of timed automata. Developed in conjunction with Uppsala University, Sweden and Aalborg University, Denmark, it consists of a Java based graphical user interface and a verification engine written in C++. It is freely available at http://www.uppaal.com/. Figure 8.4 shows the UPPAAL GUI. It enables the user to model a real time system as a network of timed finite states automata, with global, local variables, synchronization channels and clocks. The automata templates have to be entered by means of a graphical notation. Then, users have to specify the instances of the templates that are in the model and they can specify parameters that are passed to the templates.

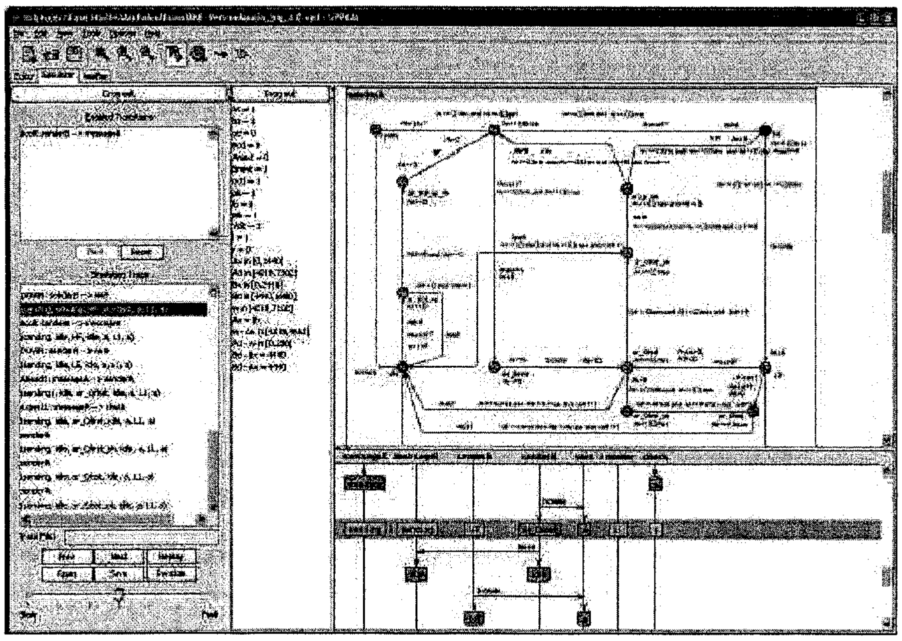


Figure 8.4: The Simulator in the UPPAAL GUI

The simulator offers the possibility to interactively run the system and check for errors introduced during modeling. The simulator shows a graphical representation of all the automata that compose the system, their current control nodes and the enabled transitions. The simulator allows the user to decide which of the enabled transitions will be executed next. Furthermore, it provides an MSC view of the execution (in the lower right part of Figure 8.4). In the simulator the user can also retrieve the values of all global and local variables and clocks. For the clocks the intervals of possible values are shown.

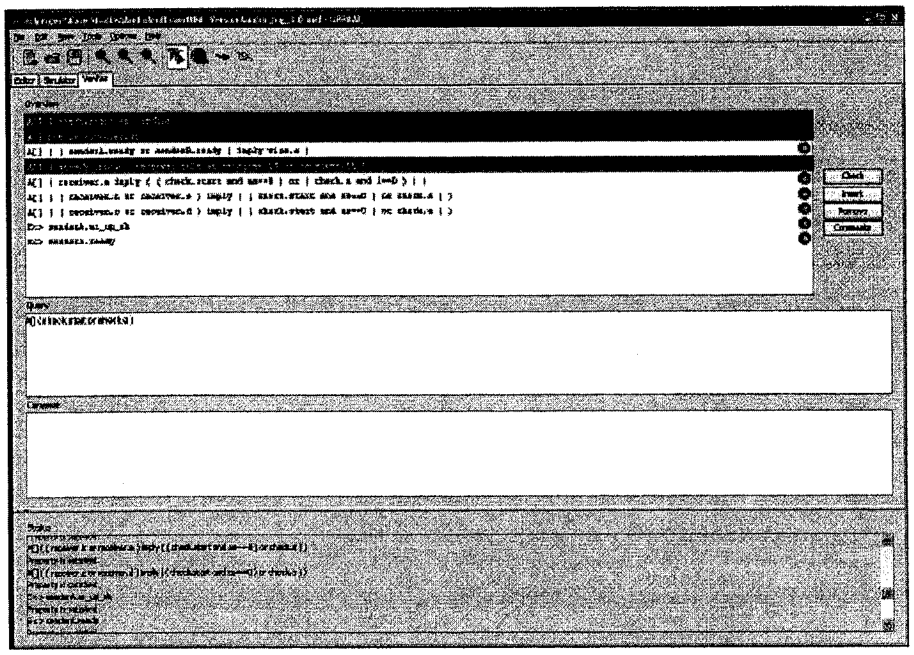


Figure 8.5: The Verifier in the UPPAAL GUI

The verification engine uses on-the-fly verification combined with symbolic techniques to overcome the state space and region space explosion problem caused by parallel composition of timed automata. UPPAAL can verify safety, bounded liveness and reachability properties on real time systems. The result of the model checking may include a diagnostic trace (showed in the simulator GUI), if the property is not satisfied. Figure 8.5 illustrates the verifier GUI.

UPPAAL Extended Timed Automata

The UPPAAL modeling language extends timed automata with the following additional features:

- **Templates** automata are defined with a set of parameters that can be of any type (e.g., int, chan). These parameters are substituted for a given argument in the process declaration.
- **Constants** are declared as *const name value.* Constants by definition cannot be modified and must have an integer value.
- **Bounded integer variables** are declared as *int[min,max] name*, where *min* and *max* are the lower and upper bound, respectively. Guards, invariants, and assignments may contain

expressions ranging over bounded integer variables. The bounds are checked upon verification and violating a bound leads to an invalid state that is discarded (at run-time). If the bounds are omitted, the default range of -32768 to 32768 is used.

- **Binary synchronization** channels are declared as *chan c*. An edge labeled with c! synchronizes with another labeled *c?*. A synchronization pair is chosen non-deterministically if several combinations are enabled.

- **Broadcast channels** are declared as broadcast *chan c*. In a broadcast synchronization one sender *c!* can synchronize with an arbitrary number of receivers *c?*. Any receiver than can synchronize in the current state must do so. If there are no receivers, then the sender can still execute the *c!* action, i.e. broadcast sending is never blocking.

- **Urgent synchronization** channels are declared by prefixing the channel declaration with the keyword urgent. Delays must not occur if a synchronization transition on an urgent channel is enabled. Edges using urgent channels for synchronization cannot have time constraints, i.e., no clock guards.

- **Urgent locations** are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location. Hence, time is not allowed to pass when the system is in an urgent location.

- **Committed locations** are even more restrictive on the execution than urgent locations. A state is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations. Main purpose of committed locations is to create atomic sequences of transitions. Committed locations reduce the state space considerably by eliminating interleaving.

- **Arrays** are allowed for clocks, channels, constants and integer variables. They are defined by appending a size to the variable name, e.g. chan c[4]; clock a[2]; int[3,5] u[7];.

- **Initializers** are used to initialize integer variables and arrays of integer variables. For instance, int i := 2; or int i[3] := {1, 2, 3};. Only integers are assigned to clocks.

UPPAAL Symbolic State Exploration

Based on timed automata semantics introduced in Section 7.5.2, it is easy to notice that the state space is uncountable. However, it is a well-known fact that timed automata have a finite-state symbolic semantics [AD94] based on countable symbolic states of the form (l,D) where $D \in \mathscr{B}(C)$:

- $\langle l,D\rangle \rightarrow \langle l, norm(M,(D \wedge I(l)) \uparrow \wedge I(l))\rangle$.

- $\langle l,D\rangle \rightarrow \langle l', r(g \wedge D \wedge I(l) \wedge I(l'))$ if $l \xrightarrow{g,r} l'$.

where $D\uparrow = \{u+d \mid u \in D \wedge d \in \mathbb{R}_+ \}$ (the future operation) and $r(D) = \{[r \mapsto 0]u \mid u \in D \}$. The function norm: $N \times \mathscr{B}(C) \rightarrow \mathscr{B}(C)$ normalizes the clock constraints with respect to the maximum constant M of the timed automaton. Normalizing the clock constraints guarantees a finite state space. We refer the reader [AD94] for in-depth treatment of the subject.

The state space exploration algorithm is shown in Figure 8.6. The waiting list (i.e. WAITING) contains unexplored but reachable symbolic states and the passed list (i.e. PASSED) contains all explored symbolic states. UPPAAL searches for states in the passed list that either cover the new state (in this case the new state is discarded) or is covered by it (in this case replaces the existing state covered by it).

```
PASSED := $\varnothing$
WAITING := $\{(l_0,D_0)\}$
**repeat**
    get (l,D) from WAITING
    **if** D $\subsetneq$ D' **for all** (l,D') $\in$ PASSED **then**
        add (l,D) to PASSED
        SUCC := { (l',D'): (l,D) $\rightarrow$ (l',D') $\wedge$ D' $\neq \varnothing$ }
        **for all** (l',D') $\in$ SUCC **do**
      put (l,D') $\in$ WAITING
        **od**
    **end if**
**until** WAITING = $\varnothing$
```

Figure 8.6: The Symbolic State Space Exploration Algorithm

UPPAAL Property Verification

Properties in UPPAAL have one of the following forms:

- A[] Expression; A<> Expression ; E<> Expression ; E[] Expression.
- Expression --→ Expression: this is 'Leads to (response)' operator. $\varphi \rightsquigarrow \psi$ is equivalent to: A[]($\varphi \Rightarrow$ A<> ψ)
- A[] not deadlock: A deadlock is a state in which no action transition will ever be enabled again. In other words:

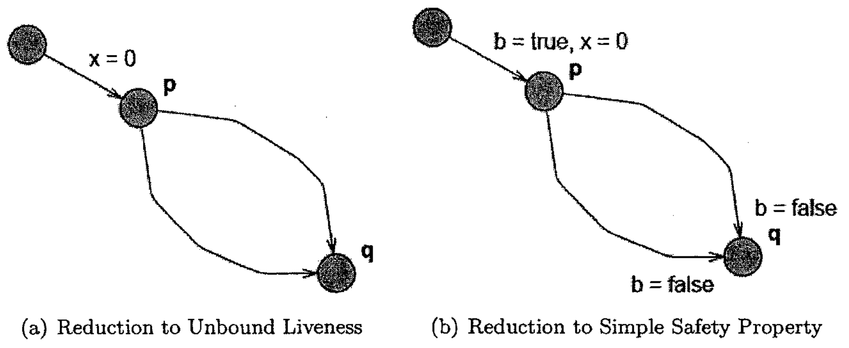
$$(l,u) \models deadlock\ iff\ \forall\ d\ \geq 0, a \in Act : (l, u+d) \overset{a}{\nrightarrow} \qquad (7)$$

Where the expressions must be type safe, side effect free, and evaluate to a boolean. Only references to integers variables, constants, clocks, and locations are allowed (and arrays of these).

Bounded Liveness Checking: Bounded liveness checking establishes that the property in question will hold within a certain upper time-limit. UPPAAL offers *time-bounded leads-to* operator $\varphi \rightsquigarrow_{\leq t} \psi$ expressing that whenever the state property φ holds then the state property ψ must hold within at most t time-units thereafter. There are three solutions to implement bounded liveness properties in UPPAAL:

1. **Reduction to unbound liveness:** The model under verification is extended with an additional clock x which is reset whenever p starts to hold. The *time-bounded leads-to* property

p $\leadsto_{\leq t}$ q is simply obtained by verifying $p \leadsto$ (q $\wedge$ x $\leq$ t). Figure 8.7(a) shows the corresponding timed automata. Note that x should not reset several times before q becomes true.



(a) Reduction to Unbound Liveness (b) Reduction to Simple Safety Property

Figure 8.7: Reduction to Unbound Liveness

2. **Reduction to simple safety property:** The model under verification is extended with a boolean variable b and an additional clock x. The boolean variable b must be initialized to *false*. Whenever p starts to hold b is set to *true* and the clock x is reset. When q commences to hold b is set to *false*. Thus the truth-value of b indicates whether there is an obligation of q to hold in the future and x measures the accumulated time since this unfulfilled obligation started. The *time-bounded leads-to* property p $\leadsto_{\leq t}$ q is simply obtained by verifying the UPPAAL safety property: A[] (b implies x $\leq$ t). Figure 8.7(b) shows the corresponding timed automata.

3. **Reduction to reachability with test automaton:** This solution is based on augmenting the model under verification with a test-automaton. The model under verification is extended with two broadcast channels a and b such that when p becomes true, the automaton sends on channel a and when q becomes true, the automaton sends on channel b. The test automaton should go to an error state when the time between a signal on a and b reaches t. Then it is sufficient to check : A[] not Test.bad. Note that this solution works even when p becomes true several times before q. Figure 8.8 shows the corresponding timed automata.

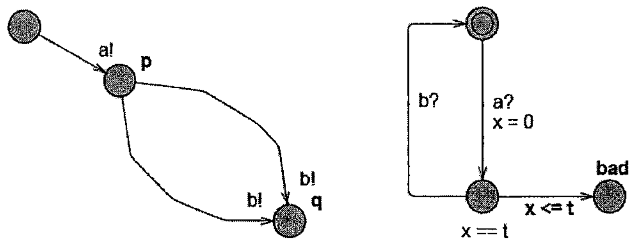


Figure 8.8: Reduction to Reachability with Test Automaton

Verification Options: UPPAAL supports many verification options:

- Breadth-first: A state space exploration method that uses a queue to implement the waiting list (see algorithm in Figure 8.6).

- Depth-first: A state space exploration method that uses a a stack to implement of the waiting list (see algorithm in Figure 8.6).

- State space reduction: For acyclic systems, a passed list is not needed to guarantee termination. However, it is useful for efficiency. Options for state space reduction include:(1) None: Store all states; (2)Conservative: Store all non-committed states; (3) Aggressive: Only symbolic states involving loop-entry points need to be stored in the passed list to guarantee termination.

- Reuse state space: When checking a property, UPPAAL searches in existing *passed* list before continuing search.

- State space representation

  - DBM

  - Compact

  - Under approximation

  - Over approximation

- Diagnostic trace

Experiments showed that breadth-first search is often much faster than depth-first search when generating the complete state space [BHV00] because depth-first search order causes higher degree of fragmentation of the zones that breadth-first order, resulting in a higher number of symbolic states being generated. We refer the reader to [LPY97] for a detailed introduction to UPPAAL.

UPPAAL Extensions

Many extensions of UPPAAL have been proposed to address validation related areas:

- *UPPAAL CORA* is a branch of UPPAAL for Cost Optimal Reachability Analysis developed by the UPPAAL team as part of the VHS and AMETIST projects. Whereas UPPAAL supports model checking of timed automata, UPPAAL CORA uses an extension of timed automata called LPTA. LPTA allows you to annotate the model with the notion of cost. This can be the cost of delay in certain situations or the cost of particular actions. UPPAAL CORA then finds optimal paths matching goal conditions.

- *UPPAAL TRON* is a testing tool, based on UPPAAL engine, suited for black-box conformance testing of timed systems, mainly targeted for embedded software commonly found in various controllers. By online we mean that tests are derived, executed and checked simultaneously while maintaining the connection to the system in real-time.

- *UPPAAL TIGA* is an extension of UPPAAL and it implements an efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties. Though timed games for long have been known to be decidable there has until now been a lack of efficient and truly on-the-fly algorithms for their analysis.
- *UPPAAL CoVer* is a tool for creating test suites from UPPAAL models with coverage specified with coverage observers (i.e., observer automata).

8.2 Early Stages Verification Approach

Figure 8.9 illustrates our proposed early stages verification approach. The approach is iterative and is composed of four iterations. First, the UCM system specification is formalized in terms of timed automata. System properties are initially defined in natural language or as UCM maps [2]. These properties are then formalized in terms of TCTL temporal logic. Next, both timed automata specification and TCTL properties are fed to UPPAAL model checker. The result is either a claim that the property is true (resulting in a valid model with respect to the checked property) or else a counter example falsifying the property. In the third iteration, the obtained counter example can be used as an input to our *'closer look'* step where UCM slicing (introduced in Section 5.2) is applied to reduce the specification size to help pinpoint the design flaw. However, choosing an *appropriate* slicing criterion based on the counter example remains a challenge. Once the error is discovered, change impact analysis is applied to assess the impact of fixing the specification. This results in fixing the specification or/and the property to be checked.

[2]chapter 9 discusses how Use Case Maps can be used as a property specification language

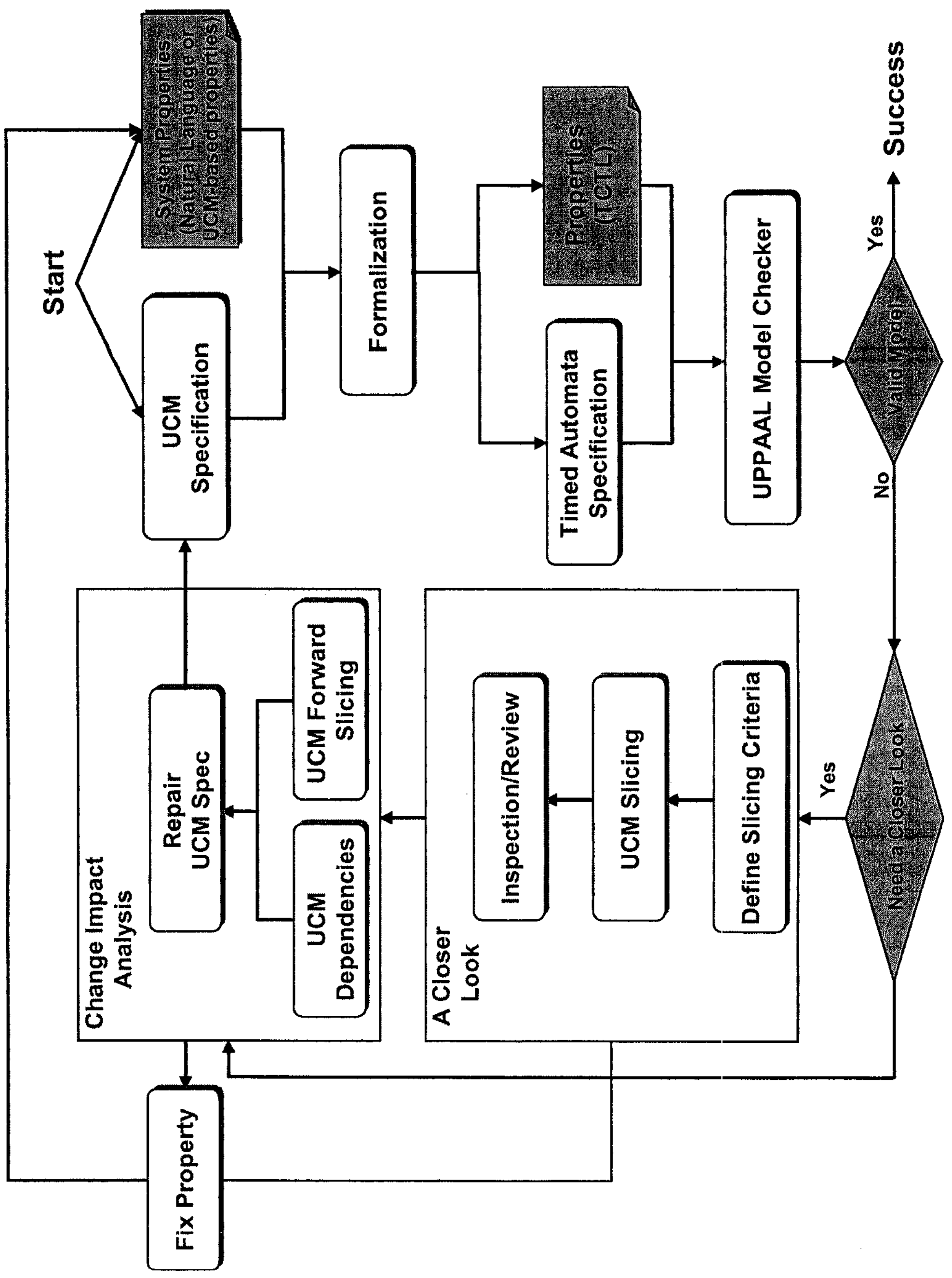


Figure 8.9: Early Stages Verification Approach

8.3 Optimizing TA Specifications

8.3.1 Sequential vs. Parallel Control Flows

The transfer of control between sequential constructs occurs in a deterministic way (i.e., in complete order), while concurrent executions result in different execution orders (i.e., partial order). Consequently, a UCM specification may be decomposed into a collection of sequential paths. For instance, the generic UCM in Figure 8.10 may be decomposed into five segments resulting in five processes, one process for AND-Fork and one process for AND-Join. Concurrent control constructs such as AND-Forks, AND-Joins and OR-Joins (in the case of merging concurrent flows) represent the glue that connects different UCM segments.

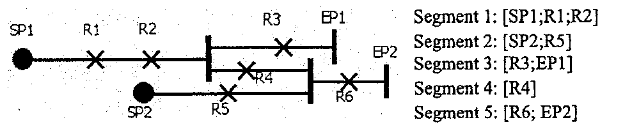


Figure 8.10: UCM Parallel Flows Decomposition

Note: A further decomposition based on UCM component bindings may be considered, but it is optional.

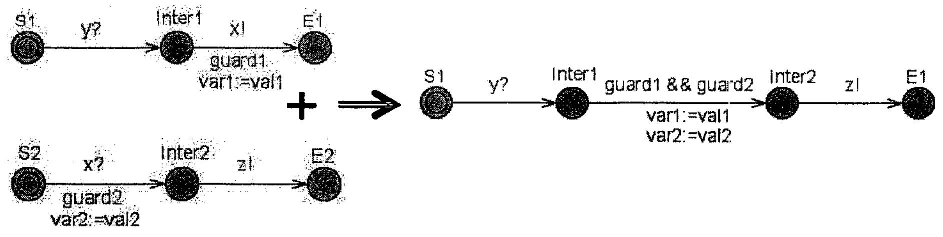
8.3.2 Sequential Composition of Timed Automata

The sequential composition of UCM TA templates is based on the resolution of all synchronizations. The transfer of control from one UCM construct to another is done through synchronization (i.e. offer(a!) and acceptance(a!)) on the channel representing the edge connecting the construct to each other. This synchronization takes place in the transitions leading to locations labeled *end*. Figure 8.11(a) illustrates such a generic sequential composition for processes having a single *end* location, while Figure 8.11(b) illustrates a sequential composition for processes having multiple *end* locations, as they typically result from the use of OR-Forks and Timers.

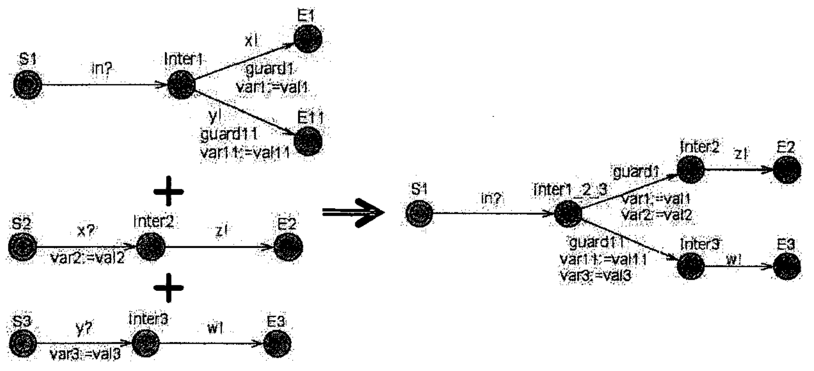
One of the challenges that we faced in building our models is UPPAAL lack of maximal progress. In the upcoming section, we present solutions to overcome this challenge.

8.4 Maximal Progress in UPPAAL

In UPPAAL, communication between two automata can be achieved through rendezvous synchronization, broadcast synchronization and/or shared variables. In rendezvous synchronization, there is a handshake between two automaton on the same channel. The transition having the sending action won't be enabled unless there is a receiver waiting on the same channel on which the sender is sending, and vice-versa. Many real-time systems need to communicate asynchronously by means of events that are triggered by change of some state variable or by time passing. UPPAAL supports



(a) Sequential Composition 1



(b) Sequential composition 2

Figure 8.11: TA Sequential Composition

the notion of *guarded transitions*, where an automaton can wait on some state-variable or time based condition to be satisfied.

In our first attempt to model edges connecting UCM constructs, we have considered the use of shared global variables. This solution is easy to implement but it does not guarantee maximal progress semantics. The use of shared variables was substituted by the use of rendezvous synchronization on entering and exiting automaton transitions (for transfer of control between processes). Invariants are assigned to intermediate states and guards are added to outgoing transitions. This solution preserves system functionalities and ensures maximal progress from a simulation perspective.

From a verification point of view, the lack of maximal progress semantics causes the UPPAAL verifier to give false positives and makes the state space larger than necessary by introducing inappropriate non-determinism, where time based transitions are also enabled apart from the shared variable based guarded transitions. This situation happens especially in stubs, plug-in start points and plug-in end points. Indeed, the fact that no time constraint is attached to plug-in start point may lead to a plug-in spending an infinite amount of time in its start location. To force UPPAAL to trigger the plug-in start point (hence achieve maximal progress), we use the concept of urgent locations (since time does not progress in an urgent location) in all stub TA locations (except the start location), plug-in start point intermediate location and plug-in end point intermediate location. One potential limitation of the use of urgent locations is that they cannot be used in conjunction with timed invariants, which in our case is not a problem since we are using them only in stubs locations, plug-in start and end points. This solution is applied while verifying the simple telephone system in Section 8.5.2.

8.5 Applying Early Stages Verification Approach

In this section, we apply the verification approach presented in the previous section on the simple telephone system (Section 2.1.4).

8.5.1 Simple Telephone System Decomposition

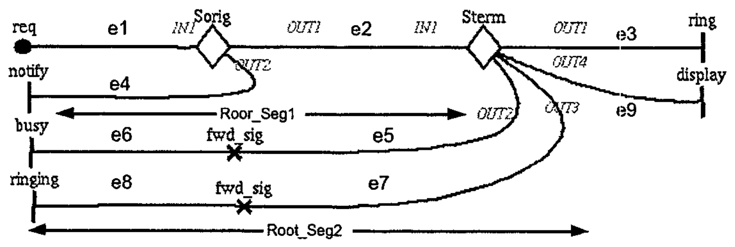
Figure 8.12 shows the decomposition of the simple telephone system. Sequential sequences of UCM constructs are grouped to form separate segments. Stubs and AND-forks are represented by single templates as implemented in Section 7.5.4.

The root map (Figure 8.12(a)) is composed of start point *req*; stubs *SOrig* and *Sterm*; end points *busy*, *ring* and *display*; and two segments *Root_Seg1* (composed of responsibility *fwd_sig* and end point *busy*) and *Root_Seg2* (composed of responsibility *fwd_sig* and end point *ringing*).

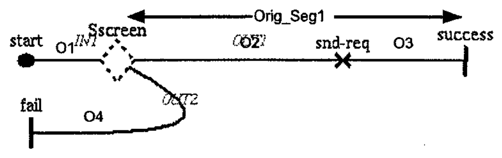
The Originating plug-in SOrig (Figure 8.12(b)) is composed of start point *start*, dynamic stub *Sscreen*, end point *fail* and segment *Orig_Seg1* which is composed of responsibility *snd-req* and end point *success*. Plug-in (Figure 8.12(c)) is represented by one single sequential segment *OCS_Seg*.

Terminating plug-in *Sterm* is composed of dynamic stub *Sdisplay*, end point *disp*, AND-Fork *AF_term* and segments *Term_Seg1*, *Term_Seg2* and *Term_Seg3*.

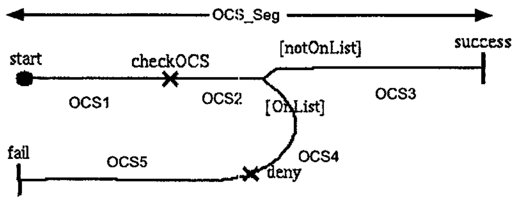
CND plug-in is composed of start point *start*, AND-Fork *AF_term*, end points *disp* and *success*, and segment *CND_Seg1*.



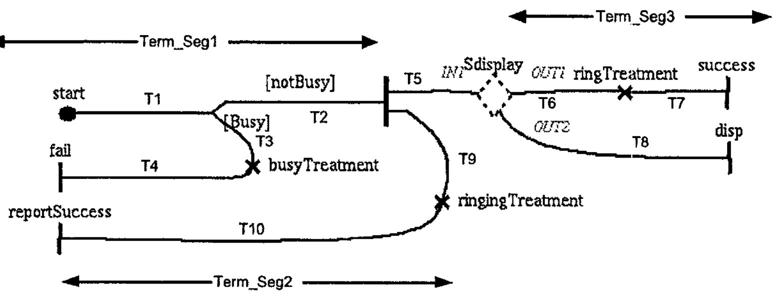
(a) Sequential Composition in Root Map



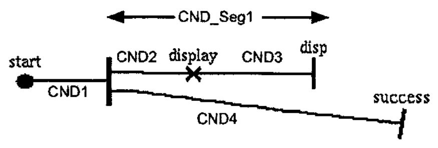
(b) Sequential Composition in Originating Plug-in (SOrig)



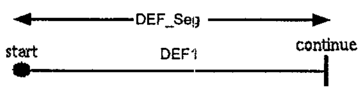
(c) Sequential Composition in OCS Plug-in



(d) Sequential Composition in Terminating Plug-in



(e) Sequential Composition in CND Plug-in

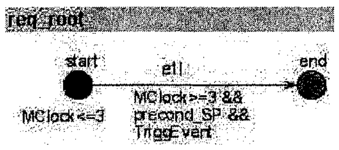


(f) Sequential Composition in Default Plug-in

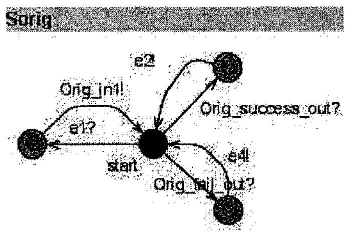
Figure 8.12: Sequential Composition of Simple Telephone System

8.5.2 Simple Telephone System: Network of Timed Automata

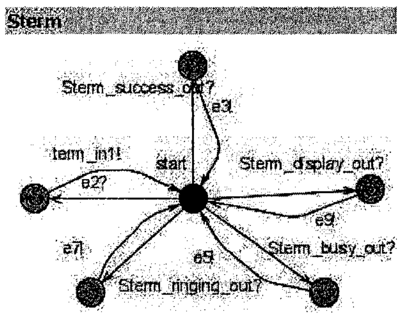
Figures 8.13,8.14,8.15,8.16 and 8.17 illustrate respectively the UPPAAL implementation of the root map, the originating plug-in, the OCS plug-in, the terminating plug-in, the CND plug-in and the default plug-in. We assign a duration between 1 and 3 to all responsibilities with a delay between 0 and 6. For instance responsibility *fwd_sigB* has a duration between 1 and 2 with delay between 1 and 4.



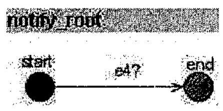
(a) Start Point req



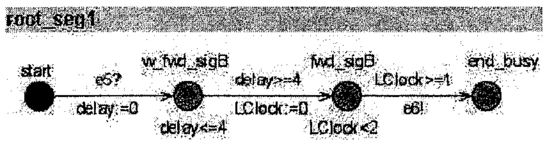
(b) Stub Sorig



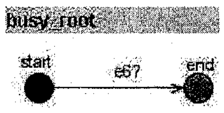
(c) Stub Sterm



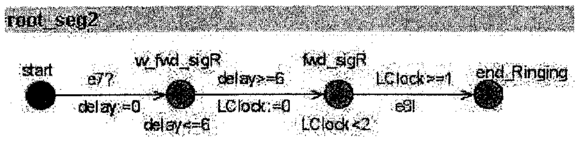
(d) End Point notify



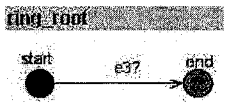
(e) Root_Seg1



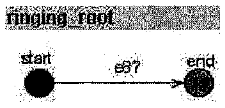
(f) End Point busy



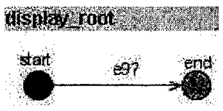
(g) Root_Seg2



(h) End Point ring

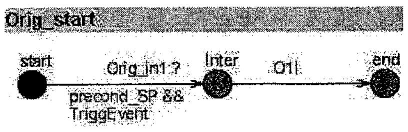


(i) End Point ringing

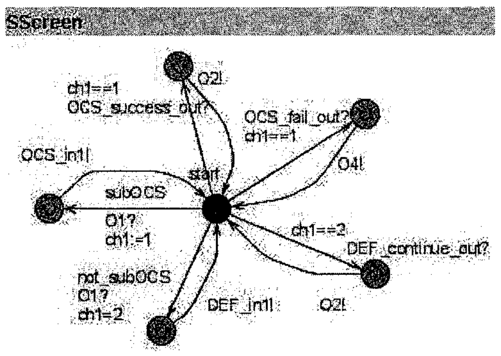


(j) End Point display

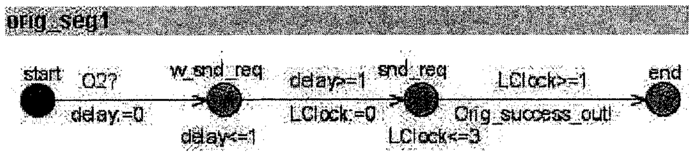
Figure 8.13: UPPAAL Implementation of the Root Map



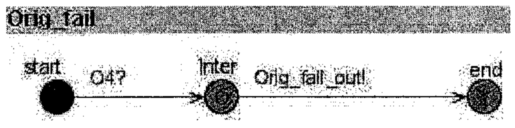
(a) Start Point:start



(b) Stub Sscreen



(c) Orig_seg1



(d) End Point: fail

Figure 8.14: UPPAAL Implementation of Originating Plug-in

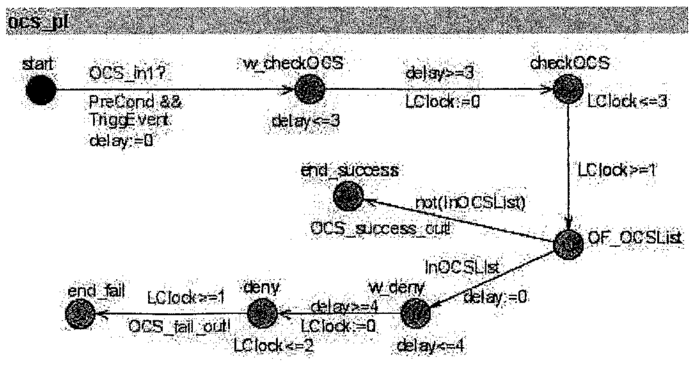
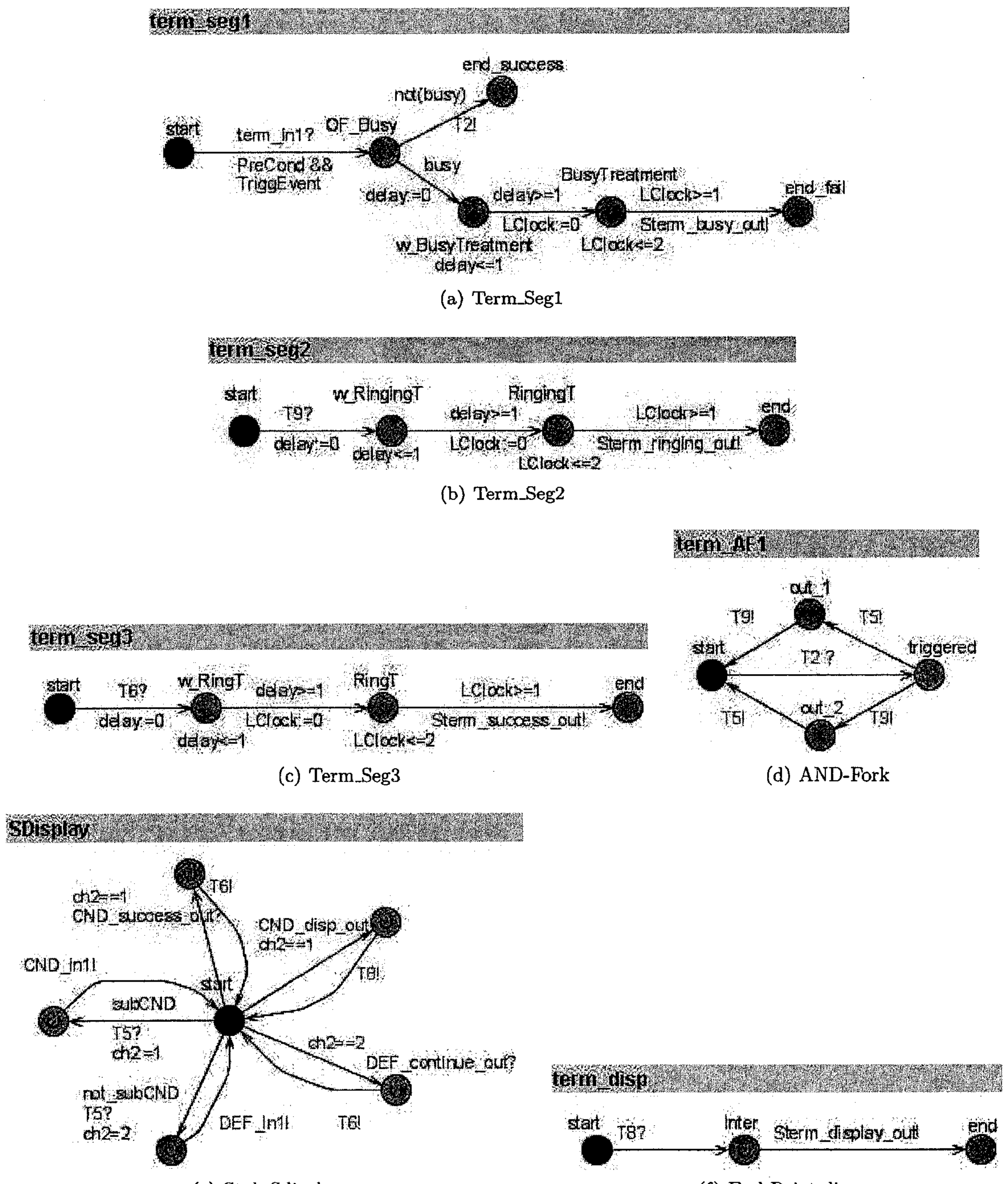


Figure 8.15: UPPAAL Implementation of OCS Plug-in



(a) Term_Seg1

(b) Term_Seg2

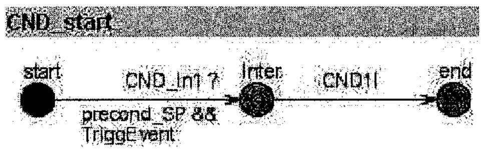
(c) Term_Seg3

(d) AND-Fork

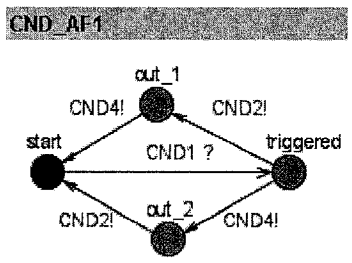
(e) Stub Sdisplay

(f) End Point disp

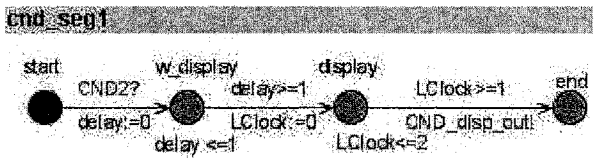
Figure 8.16: UPPAAL Implementation of the Terminating Plug-in



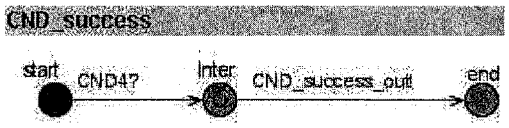
(a) Start Point: start



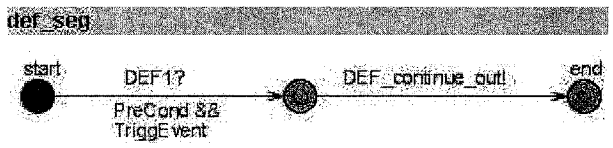
(b) AND-Fork



(c) CND_Seg1



(d) End point: success



(e) DEF_Seg

Figure 8.17: UPPAAL Implementation of CND and DEFAULT Plug-in Maps

8.5.3 Simple Telephone System: Property Verification

In this section, we verify selected properties against the simple telephone system implemented in Section 8.5.2.

- <u>**Safety Properties:**</u>

  - **Property 1:** The call is allowed to proceed only when the caller in not on OCS list (i.e. InOCSList=false). This property is translated into the following UPPAAL formula:

    $$\mathbf{A}[\,](\mathit{ocs\_pl.end\_success})\ \mathbf{imply}\ (!\mathit{InOCSList}) \tag{8}$$

    This property is checked to be true by the UPPAAL verifier.

  - **Property 2:** The caller is subscribed to OCS and the callee is subscribed to CND. The caller's number is displayed (i.e. responsibility *display*) only when the callee is not busy. This property is translated into the following UPPAAL formula:

    $$\mathbf{A}[\,](\mathit{cnd\_seg1.display})\ \mathbf{imply}\ (!\mathit{InOCSList}\ \mathbf{and}\ !\mathit{busy}) \tag{9}$$

    This property is checked to be true by the UPPAAL verifier.

- <u>**Liveness Properties:**</u>

  - **Property 3:** When both OCS and CND are enabled and active (i.e. InOCSList = false, subCND=true and busy=true), the caller should receive a ringing tone. This property is translated into the following UPPAAL formula:

    $$\mathbf{A}{<>}(\mathit{ringing\_root.end}\ \mathbf{imply}\ (!\mathit{InOCSList}\ \mathbf{and}\ !\mathit{busy}\ \mathbf{and}\ \mathit{subCND})) \tag{10}$$

    This property is checked to be true by the UPPAAL verifier.

  - **Property 4:** Both OCS and CND are enabled and active. When the caller is busy a *busy-tone* is sent to the caller (responsibility *root_seg1.fwd_sigB*). This property is translated into the following UPPAAL formula:

    $$(\mathit{subOCS}\ \mathbf{and}\ !\mathit{InOCSList}\ \mathbf{and}\ \mathit{subCND}\ \mathbf{and}\ \mathit{busy}) \dashrightarrow \mathit{root\_seg1.fwd\_sigB} \tag{11}$$

    This property is checked to be true by the UPPAAL verifier.

- <u>**Response Properties:**</u>

  - **Property 5:** When the callee is in OCSList, the deny operation is followed by a notification sent to the caller. This property is translated into the following UPPAAL formula:

    $$(\mathit{subOCS}\ \mathbf{and}\ \mathit{InOCSList}\ \mathbf{and}\ \mathit{ocs\_pl.deny}) \dashrightarrow \mathit{notify\_root.end} \tag{12}$$

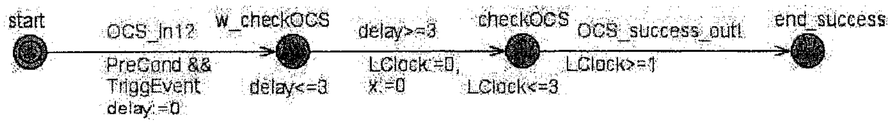
    The property was verified by UPPAAL as expected.

- **Bounded Liveness Property:**

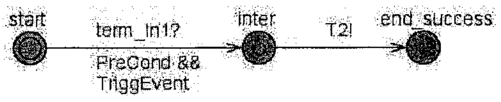
  - **Property 6:** No more than 7 time units elapse between the OCS checking (i.e. responsibility *checkOCS*) and the display operation (i.e. responsibility *display*). As explained in Section 8.1.5 bounded liveness properties cannot be expressed directly in the logical property language of UPPAAL without adding additional clocks. We introduce clock x to measure the time between *checkOCS* and *display.* Clock x is reset before entering location *checkOCS* and it is read at location *display* part of CND plug-in map. This property is translated into the following UPPAAL formula:

$$!InOCSList \textbf{ and } (ocs\_pl.checkOCS) \dashrightarrow (cnd\_seg1.display \textbf{ and } x \leq 7) \qquad (13)$$

This property was not satisfied by the model and the diagnostic trace shows that x should be greater than 12 when reaching display operation. The trace (shortest trace) starts at the start point *req_root* and ends at root map end point *ringing_root.* In order to investigate the reason of this failure and pinpoint where the delay was introduced, the specification is sliced according to slicing criterion *ringing_root.* We obtain the same UCM slice as the one presented in Figure 5.12. The resulting UPPAAL processes (as a result of the slicing) are shown in Figure 8.18. The OCS plug-in map is reduced to 4 states from 8 states, term_seg1 is reduced to 3 states from 6 states, processes orig_fail, notify_root and term_busyTreatment are sliced out. This reduced UPPAAL specification allows for a reduced state space. Hence, more efficient verification.



(a) Sliced OCS Plug-in Map



(b) Sliced Term_Seg1

Figure 8.18: Sliced UPPAAL Implementation of OCS and Terminating Plug-in Maps

From the analysis of the reduced UPPAAL specification one can observe that it is necessary to reduce the duration of responsibility *checkOCS* to 2 from an initally maximum value of 3, to satisfy the property.

8.6 Chapter Summary

In the beginning of this chapter, we have provided an introduction to model checking, temporal logics formalisms (standard temporal logic formalisms such as CTL*/CTL/LTL and real-time temporal logic such as MTL and TCTL) and supporting tools. We have devoted a full section to UPPAAL model checker which is our selected verification tool. Then we have discussed the problem of state space explosion and provided an overview of existing techniques to cope with it. The core of this chapter consists on the combination of model checking technique with UCM requirement slicing (introduced in Chapter 5) to formally verify timed Use Case Maps specifications. The UPPAAL model is derived manually from the UCM specification according to the TA templates presented in Chapter 7. To be able to efficiently verify UCM models, we have addressed UPPAAL's lack of maximal progress issue and we have proposed a mechanism to sequentially compose timed automata. Finally, we have applied our approach to verify some properties of the simple telephone system.

The automatic generation of UPPAAL model from UCM specification is left for future work. Furthermore, as discussed in Chapter 5, the choice of the appropriate slicing criterion to reduce the UCM specification (for uncovering design flaws and change impact analysis) remains a challenge that needs to be addressed properly.

Chapter 9

Use Case Maps as A Property Specification Language

Although there exists a significant body of research in the area of formal verification and model checking tools of software and hardware systems, there has been only a limited industry and end-user acceptance of these tools. Besides the technical problem of state space explosion discussed in the previous chapter, one of the main reasons for this limited acceptance is the unfamiliarity of users with the required specification notation. Requirements have to be typically expressed as temporal logic formalisms and notations. Property specification patterns were successfully introduced to bridge this gap between users and model checking tools. They enable also non-experts to write formal specifications that can be used for automatic model checking. In this chapter[1], we propose an abstract high level pattern-based approach to the description of property specifications based on Use Case Maps. We present a set of commonly used properties with their specifications that are described in terms of occurrence, ordering and temporal scopes of actions. Furthermore, our approach also supports the description of properties with respect to their architectural scope. We provide a mapping of our UCM property specification patterns in terms of CTL [CES86], TCTL [Alu92] and ArTCTL (Architectural TCTL), an extension to TCTL, introduced in this research that provides temporal logics with architectural scopes.

9.1 Introduction

Model checking has been widely used as a method to formally verify finite-state concurrent systems, such as communication protocols. System properties are expressed as temporal logic formulas, and efficient algorithms are used to traverse the resulting model to check whether the system is consistent with the specified properties. Many temporal logics, such as linear-time temporal logic (LTL) [MP92], computational tree logic (CTL) [CES86] and ACTL [NV90] have been suggested as formal languages for property specifications. However, the use of temporal logics is still limited

[1]This chapter content is published in the Journal of Software and System Modeling (SoSyM) [HRD07b]

to users with a good mathematical background because temporal logic formulae are difficult to understand and even more difficult to create. To bridge this gap between practitioners and model checking tools, many authors have proposed property specification patterns [ABKO04, DAC99, GL05, KC05a, KC05b] to guide users in expressing system requirements directly in temporal logic.

Previously published pattern systems vary from simple specification patterns dealing with occurrences of events or states (describing *what* must occur) and scopes (describing *when* the pattern must hold) [DAC99], to real-time pattern properties considering information about time [GL05, KC05b, TYZP05]. However, to the best of our knowledge, the existing pattern systems deal mainly with behavioral aspects of systems but fail to capture the architectural scope of a system (describing *where* the pattern must occur). Applying an architectural scope allows to describe architecture related issues, like "*action P is executed in component C*". Building a property pattern system that considers functional, timing and architectural aspects all together will improve the verification of distributed real-time embedded systems. Such systems often are based on an heterogeneous system architecture; they consist of components that range from fully programmable processor cores to fully dedicated hardware components for time-critical application tasks.

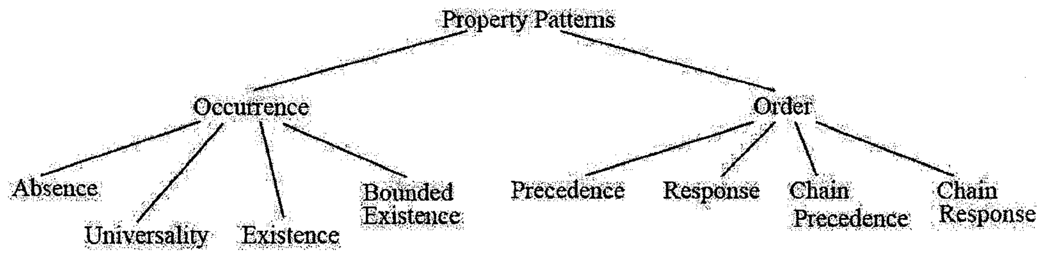


Figure 9.1: Pattern Hierarchy by Dwyer et al. [DAC99]

Our research builds upon previous work on property patterns introduced in [DAC99, DAC98, GL05]. We propose an abstract high level pattern-based approach that supports the description of property specifications using Use Case Maps language (UCM).

In what follows, we present a novel approach that addresses the following concrete issues:

- The Use Case Maps language was extended to simplify the writing and understanding of properties, by providing a UCM property pattern system with templates that explicitly capture functional, timing and architectural property patterns. The proposed UCM property patterns system offers users a visual and an easy to learn framework for the specification of complex properties without the use of textual temporal logic formalisms.

- Having both, the requirement specification and properties described using the same formalism (i.e. Use Case Maps) will allow for a more detailed analysis while preserving a high level of abstraction.

- A mappings for the UCM property specification patterns in terms of CTL and $TCTL$ temporal logics is provided.

- We provide an extension to the well-known *TCTL* [Alu92] temporal logic formalism by including additional architectural constraints. The proposed extension is named *ArTCTL* (Architectural TCTL). However, the definition of a general formal framework for architectural temporal logic is left for future work.

9.2 Specification Patterns

In this section, we overview the specification patterns by Dwyer et al. [DAC99, DAC98], the timed property patterns by Gruhn et al. [GL05] and the real-time property pattern by Konrad et al. [KC05b].

9.2.1 Untimed Specification Patterns

In [DAC99], Dwyer et al. collected over 500 specifications from several sources and observed that nearly all the properties could be classified into a hierarchy of basic patterns based on their semantics. This hierarchy, illustrated in Figure 9.1, distinguishes properties that deal with the occurrence and ordering of states/events during a system execution. Each of these patterns describes an intent (the structure of the specified behavior), a scope (the extent of program execution over which the pattern must hold), mappings into some specification formalisms for finite-state verification tools (LTL [MP92], CTL [CES86], QRE [OO90]), some known uses, and relationships to other patterns. For instance, the intent of the Precedence pattern is a relationship between a pair of events/states where the occurrence of the first is a necessary precondition for the occurrence of the second (also known as *Enables*).

In what follows we describe briefly the property patterns and their scope as introduced by Dwyer's et al. A more detailed description of these patterns can be found in [DAC99].

- **Absence**. A given event/state P does never occur within a scope.
- **Universality**. A given event/state P occurs throughout a scope.
- **Existence**. A given event/state P must occur within a scope.
- **Bounded Existence**. A given event/state P must occur at least/exactly or at most k times within a scope.
- **Precedence**. An event/state P must always be preceded by an event/state Q within a scope.
- **Response**. An event/state P must always be followed by an event/state Q within a scope.
- **Chain Precedence/Chain Response**. A sequence of events or states P_1, …,P_n must always be preceded/followed by a sequence of events/states Q_1, …, Q_n within a scope.

Dwyer et al. identified five scopes, or segments of system execution:

- **Global**. The pattern must hold during the complete system execution.

- **Before.** The pattern must hold up to a given event/state Q.
- **After.** The pattern must hold after the occurrence of a given event/state Q.
- **Between.** The pattern must hold from the occurrence of a given event/state Q to the occurrence of a given event/state R.
- **After-Until.** Like *between*, but the designated part of the execution continues even if the second event/state R does not occur.

The pattern catalog allows for reasoning about occurrence and order of events. However, it does not support quantitative reasoning about time due to the fact that real-time properties cannot be specified using these existing patterns. In Dwyer's pattern system, properties like 'P must always be followed by Q within k time units' cannot be expressed. In the following section, we present an overview of the work of Gruhn et al. [GL05] and Konrad et al. [KC05b] who addressed this shortcoming. We also survey some UML-based approaches for property description.

9.2.2 Timed Specification Patterns

Konrad et al. [KC05b] have proposed real-time specification patterns that can be classified into three categories of real-time properties: duration (captures properties that can be used to place bounds on the duration of an occurrence), periodic (describes properties that address periodic occurrences), and real-time order (captures properties that place time bounds on the order of two occurrences). Figure 9.2 illustrates this pattern classification.

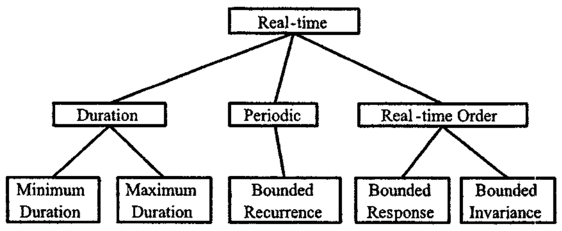


Figure 9.2: Real-Time Specification Patterns by Konrad et al.

The authors have also provided a pattern description template similar to the one proposed in [DAC99] consisting of a pattern name and classification, a pattern intent, a mapping to timed temporal logics (i.e., MTL [AH90, Koy90], TCTL [Alu92] and RTGIL [AH92]), examples of known uses, relationships and a structured English specification. The structured English specification captures the scope (*globally*, *before*, *after*, *between*, or *after-until*) followed by the type (*qualitative* or *real-time*) then the category (*duration*, *periodic*, or *real-time order* for real-time properties, and for quality properties (*occurrence* or *order*) of the property. An example of such an English description is: "Globally, it is always the case that if P holds, then S holds after at most c time unit(s)". Obtaining such description is the result of the execution of six rules (e.g., property, scope, specification, real-time Type, real-time order category and bounded response pattern).

In [GL05], Gruhn et al. proposed another catalog of patterns for real-time requirements. For each pattern, a timed observer automaton is constructed to describe the desired behavior. The observer runs in parallel with the model under verification. The observer reaches an *Error* state if and only if the property can be violated. Therefore, in order to prove that a property holds, it is sufficient to check that the observer cannot reach some location. The catalog adds the notion of time constraints to the patterns introduced by Dwyer et al. to be able to specify properties like: "Starting from the current point of time, P must occur within k time-units". The corresponding automaton is illustrated in Figure 9.3. The catalog covers many interesting timed patterns and proposes their corresponding timed automata. The use of temporal automata for specifying temporal properties have also been used by several authors [BF99, OK01].

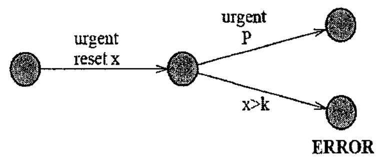


Figure 9.3: Timed Automaton for Time-bounded Existence

A similar observer concept is used in [ABKO04], where Alfonso et al. introduced VTS, a visual language to define complex event-based requirements such as freshness, bounded response, event correlation, etc., and a tool that translates these requirements into the input language of the model checker KRONOS. The user has to graphically describe the scenarios violating the requirements, which is in our opinion a major drawback. Indeed, deriving all possible scenarios that violate a given requirement is an error prone activity and the resulting set of scenarios may be incomplete. Tsai et al. [TYZP05] describe a testing approach based on scenarios and verification/robustness patterns (SP, VP/RBP). These are temporal patterns (or cause-effect relations) that allow the specification of pre- and post-conditions as well as timing constraints (e.g. optional timeout, time slices, ...), and are expressed both visually and in LTL temporal logic.

Several approaches for describing properties with UML models have been proposed. These approaches either extend OCL for temporal constraints specification or express behavioral real-time constraints in UML diagrams. Ramakrishnan et al. [RM99] extend the OCL syntax by additional grammar rules with unary and binary future oriented temporal operators (e.g., always and never) to specify safety and liveness properties. Flake and Muller. [FM02a] have developed a temporal OCL extension that enables modelers to specify state-oriented real-time constraints. This extension covers the consecutiveness of states and state transitions as well as time-bounded constraints.

Schäfer et al. [SKM01] describe systems using UML state machines and use UML collaboration diagrams to specify properties. In order to verify properties using model checker SPIN, state machines model is compiled into a PROMELA model while collaborations are compiled into sets of Büchi automata (i.e *"never claims"*). Graf et al. [OGO04] proposed a mapping of UML models into a framework of communicating extended timed automata (stereotyped as observers) to serve as property specification language. Although, these UML models have the advantage to be simpler and easier to use for experienced UML users, they suffer from the same drawback as other observer-based approaches. The models require for example the user to describe manually all the scenarios violating the requirements.

9.3 Use Case Maps Property Pattern System

In this section, we present a graphical specification pattern catalog based on the Use Case Maps notation. Our proposed pattern system covers all qualitative specification patterns introduced by Dwyer et al. [DAC99] as well as real-time specification patterns presented in [ABKO04, GL05, KC05b]. The research is motivated by the goal to capture both qualitative properties and quantitative timing requirements. Furthermore, as the structural aspects of a system can be captured without the user having to be familiar with temporal logic for the representation of the properties and the description of scenarios that violate the requirements [ABKO04, GL05].

Although, Use Case Maps is primarily a functional description language (i.e., behavior oriented), it can be used to reason about atomic propositions as well. In addition to the UCM representation, we provide a mapping of our pattern catalog to temporal logics CTL and TCTL. When reasoning about responsibilities/actions, our UCM-based pattern system can be easily mapped to ACTL [NV90], which extends CTL with actions. Like CTL, ACTL is a propositional branching-time temporal logic. While CTL is interpreted over Kripke structures, ACTL is interpreted over labeled transition systems (LTSs). A more detailed description of the relationship between CTL and ACTL can be found in [NFGR92]. Real-time properties may also be mapped to a real-time version of ACTL called ATCTL [JW02].

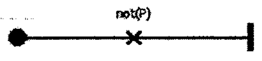
In the context of our research, one important aspect for us was not to have to extend the existing UCM language by introducing additional new notations. Instead, we extend the use of existing UCM notations. For example we extended the use of UCM labels that are typically applied to identify different UCM constructs (e.g. construct's name), by existential and universal quantifiers. These quantifiers can be then applied to specify the scope of our specification patterns.

9.3.1 Patterns

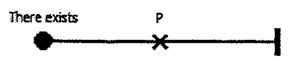
In this section, we describe the qualitative properties of the patterns introduced by Dwyer et al. [DAC99] using an UCM based representation and their mapping to CTL logic. For clarity purpose we will use a "*global scope*" to represent these properties. Temporal scopes will be discussed in Section 9.3.2.

- **Absence.** In order to describe that a given responsibility/event P never occurs within a defined scope, we extend both the UCM responsibility labels with the negation operator *not(P)*, which represent any sequence of responsibilities not containing P. The label "not($P_1, \dots, P_n$)" denotes any sequence of responsibilities that does not contain the set of responsibilities P_1, $\dots$,P_n. Figure 9.4(a) illustrates this absence property.

$$Mapping\ to\ CTL: AG(\neg P) \qquad (14)$$



(a) Absence Property



(b) Universality Property

Figure 9.4: Absence and Universality

- **Universality.** Universality is a dual of the absence property stating that a given responsibility/event P occurs (Figure 9.4(b)). Adding, an existential quantifier (i.e., there exists) to the start point label shows that responsibility/event P should occur at least once during a possible execution (e.g., at least one path).

$$Mapping\ to\ CTL: AG(P) \tag{15}$$

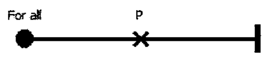
- **Existence.** The start point is labeled with the universal quantifier to state that for all possible execution paths P must occur (Figure 9.5(a)).

$$Mapping\ to\ CTL: AF(P) \tag{16}$$

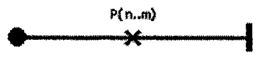
- **Bounded Existence.** Responsibility P must occur at least/exactly/most k times. This is achieved by adding cardinalities to the responsibility label. "P(n ... m)" denotes that P is repeated at least n times and at most m times (Figure 9.5(b)).

  One instance of the bounded existence pattern, where P occurs at most 2 times, is represented by the following CTL formula:

$$Mapping\ to\ CTL: \neg EF(\neg P \wedge EX(P \wedge EF(\neg P \wedge EX(P \wedge EF(\neg P \wedge EX(P)))))) \tag{17}$$



(a) Existence Property



(b) Bounded Existence

Figure 9.5: Existence and Bounded Existence

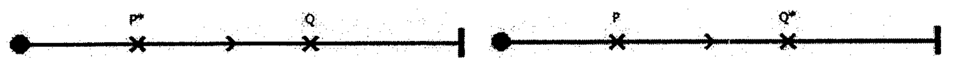
- **Response.** A directed arrow between responsibilities P and Q shows that when P occurs then an occurrence of Q should follow. The star in front of the responsibility label means: "*if P occurs*" (Figure 9.6(a)).

$$Mapping\ to\ CTL: AG(P \Rightarrow AF(Q)) \tag{18}$$

Causality is always defined by construction in UCMs. However, the arrow is added to distinguish the general response property from a restricted response property (i.e., Q should immediately follow P). In the later, the directed arrow is omitted.

- **Precedence.** The precedence property represents a restriction of the response property in the sense that Q can only follow P (Figure 9.6(b)). The star in front of the responsibility label means *"if Q occurs"*.

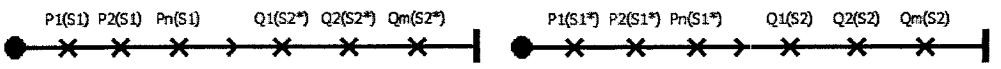
$$\textit{Mapping to CTL}: \neg E[\neg P\ U(Q \wedge \neg P)] \tag{19}$$



(a) Response Property (b) Precedence Property

Figure 9.6: Response and Precedence

- **Chain Precedence/Chain Response.** A sequence of responsibilities $P_1, \ldots, P_n$ must always be preceded/followed by a sequence of responsibilities $Q_1, \ldots, Q_n$. In addition to its name, each responsibility is labeled with the name of the chain it belongs to. Figures 9.7(a) and 9.7(b) illustrate the chain Precedence/Response. $P_1, P_2, \ldots, P_n$ belong to chain *S1* while $Q_1, Q_2, \ldots, Q_n$ belong to chain *S2*.



(a) Chain Precedence (b) Chain Response

Figure 9.7: Chain Precedence and Chain Response

Note: The chain precedence/response makes only sense for cases with non overlapping chains.

The CTL mapping of the precedence chain: P_1 precedes Q_1 and Q_2, is as follows:

$$\neg E[\neg P_1\ U(Q_1 \wedge \neg P_1 \wedge EX(EF(Q_2)))] \tag{20}$$

The CTL mapping of the response chain: Q_1, Q_2 responds to P_1, is as follows:

$$AG(P_1 \Rightarrow AF(Q_1 \wedge AX(AF(Q_2)))) \tag{21}$$

We now have introduced the elements necessary to describe a more complex requirement: **Separated Response.** Describes that a responsibility P is followed by two responses Q and

R, which are not separated by S. An *AND-Fork* is used to specify that Q and R may occur in any order (Figure 9.8).

$$\textit{Mapping to CTL}: AG(P \Rightarrow AF(Q \wedge \neg S \wedge AX(A[\neg S\ U\ R]))) \tag{22}$$

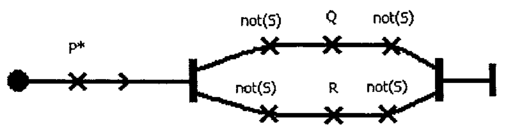


Figure 9.8: Separated Responses

9.3.2 Specification Pattern Scopes

Temporal Scopes

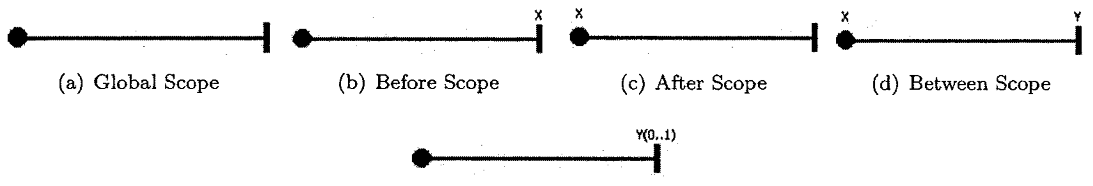
The optional temporal scopes define when the above patterns must hold. The scope is determined by specifying a start and an end state/event for the pattern. Dwyer et al. [DAC99] defined five different types of scope. For each temporal scope, we present only the mapping of the precedence property in terms of CTL. For a complete CTL mapping of the qualitative properties with respect to all the described scopes, we refer the reader to [SAn07].

- **Global**: Start and end point labels are left blank to state that the pattern must hold during the complete system execution (Figure 9.9(a)). The CTL Mapping for "S precedes P" is given by:

$$A[\neg PWS] \tag{23}$$

- **Before**: The end point is labeled with the event X to state that the pattern must hold up to a responsibility/event X (Figure 9.9(b)). The CTL mapping for "S preceded P Before R" is:

$$A[(\neg P \vee AG(\neg R))W(S \vee R)] \tag{24}$$



(a) Global Scope (b) Before Scope (c) After Scope (d) Between Scope

(e) Until Scope

Figure 9.9: Temporal Scopes

- **After**: The start point is labeled with the event X to describe that the pattern must hold after the occurrence of a responsibility/event X (Figure 9.9(c)). The CTL mapping for "S preceded P After Q" is:

$$A[\neg QW(Q \wedge A[\neg PWS])] \tag{25}$$

- **Between**: The start point is labeled with X and the end point is labeled with Y to describe that the pattern must hold from the occurrence of X to the occurrence of Y (Figure 9.9(d)). The CTL mapping for "S precedes P Between Q and R":

$$AG(Q \wedge \neg R \Rightarrow A[(\neg P \vee AG(\neg R))W(S \vee R)]) \tag{26}$$

- **Until**: The same as "*between*", but the pattern must hold even if Y never occurs. The end point is labeled with Y having a cardinality of either 0 (in case Y never occurs) or 1 (in case Y occurs) (Figure 9.9(e)). The CTL mapping for 'S preceded P After Q until R':

$$AG(Q \wedge \neg R \Rightarrow A[\neg PW(S \vee R)]) \tag{27}$$

Note: A scope label may coexist with a pattern related label on a start point. For instance, the start point of a UCM describing "an existence property that should hold after the occurrence of an event X", is labeled with "There exists—X".

Examples of Timing Requirements

- **Bounded Response:** Figure 9.10(a) describes a bounded response, where P is followed by Q after 10 time units and neither $R1$ nor $R2$ should occur between P and Q. Its corresponding TCTL mapping is:

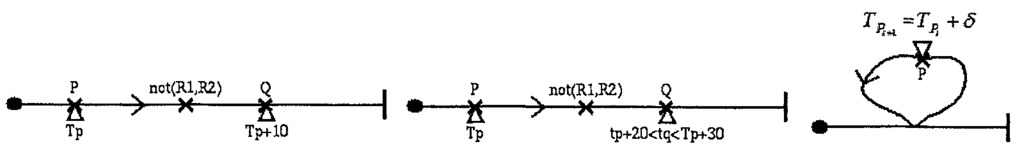
$$AG(P \rightarrow (AF_{\leq 10}Q) \wedge \neg R1 \wedge \neg R2) \tag{28}$$

  Figure 9.10(b) describes the same property but with a relaxed interval for responsibility Q. Q is supposed to occur 20 TU after the occurrence of P but not more later than 30 TU. Its corresponding TCTL mapping is:

$$AG(P \rightarrow (AF_{[20,30]}Q) \wedge \neg R1 \wedge \neg R2) \tag{29}$$

- **Periodic Recurrence:** Figure 9.10(c) illustrates a periodic occurrence of responsibility P where P occurs every δTU.

$$Mapping\ to\ TCTL : AG(AF_{\leq \delta}P) \tag{30}$$



(a) Bounded Response (b) Bounded Response with interval (c) Periodic Recurrence

Figure 9.10: Examples of Timing Requirements

Architectural Scopes

Architectural descriptions are playing an increasingly important role in the ability of software engineers to describe and comprehend software systems. Architecture is generally considered to consist of components and the connectors (interactions) between them. Architectural reasoning needs to cope with evolving system requirements, where systems evolve to migrate to new technologies or/and to include new features. These changes may modify the assumptions on which system functionalities are based. Therefore, test engineers may want to:

- Ensure that the desired topology is preserved for a specific feature (e.g., feature functionalities should be bound to a specific topology).
- Ensure that components that are intended to interact can indeed do so (e.g., there exists a scenario that is divided into many components).

In an effort to address these architectural issues, we introduce architectural scopes with the goal to increase the understandability and reasoning about architectural designs. At the same time we allow for improved analysis and testing while preserving a high level of abstraction. Use Case Maps have the benefit of integrating both behavioral and architectural aspects in one representation.

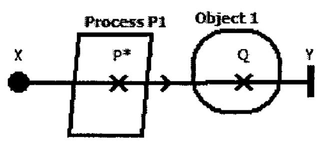
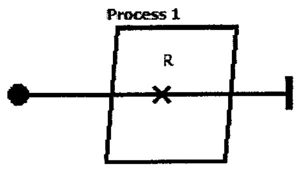


Figure 9.11: Three Scopes: Occurrence, Temporal and Architecture

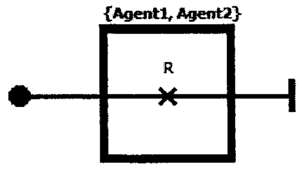
A user may for example want to express a response property, where Q should follow the occurrence of P and this should happen between any occurrence of X and Y. In addition, the property should hold only and only if the responsibility/event P is executed by *Process1* while responsibility/event Q is executed within *Object1*. This generic response property can be described as shown in Figure 9.11.

We can define five distinctive architectural scopes:

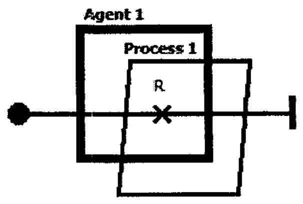
- **Component Specific:** The pattern must take place within a pre-defined component. The architectural property is violated when the responsibility/event occurs within a different component. Figure 9.12(a) illustrates a generic property where responsibility R should occur as a part of process "Process 1".



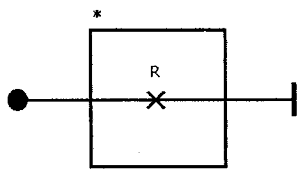
(a) Component Specific



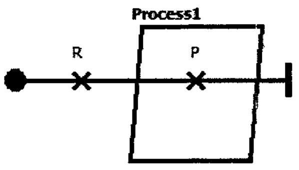
(b) Multiple Same Type Components



(c) Multiple Different Type Components



(d) Any Component



(e) Unbound

Figure 9.12: Architectural Scopes

- **Multiple Same Type Components:** The *Component Specific* scope is relaxed to give the user the possibility to specify more than one component of the same type for a certain event/responsibility. Figure 9.12(b) illustrates a generic property where responsibility R should occur as a part of either *Agent1* or *Agent2.*

- **Multiple Different Type Components:** The *Component Specific* scope is relaxed to give the user the possibility to specify more than one component of different types for a certain event/responsibility. Figure 9.12(c) illustrates such a generic property where responsibility R should occur as a part of either agent *Agent1* or process *Process1.*

- **Any Component:** The property may occur within any component of a predefined type. This is described by using "*" as the name of the component. Figure 9.12(d) illustrates a generic property, where responsibility R should occur as a part of an actor. The actor name in this case is not specified.

- **Unbound:** For unbound event/responsibility (i.e., not attached to any component), the component name or type are not relevant. The event/responsibility can take place within any component of any type. The focus is on the behavior and timing aspects rather than the architectural aspect. Figure 9.12(e) illustrates a generic property, where responsibility R is not attached to any other component.

Note: A component may be part of another component (For instance a process can fork to have childs). This architectural containment dependency may be represented as part of the property definition. Figure 9.13 illustrates a property with a responsibility R performed by *object1* which is part of process *Process1.*

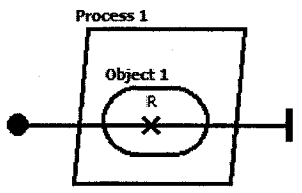


Figure 9.13: Architectural Containment Dependency

In the following Section, we give a general overview on how to extend real-time temporal logics with architectural aspects. We extend TCTL, one variant of real-time temporal logic, to include architectural constraints. We describe the formal syntax and semantics of what we name *ArTCTL* (Architectural TCTL).

9.4 Architectural Real-time Temporal Logic

Labeled transition systems (LTSs) are used to reason about non-real-time systems. For real-time systems, timed transition systems (TTS) are used, which can be seen as an extension of labeled transition systems. The passing of time is modeled by labeling transitions with non negative real numbers. The semantics of timed automata [AD94] are usually described in terms of timed transition system (TTS).

We slightly modify the classical definitions related to TTS and TA by adding the architectural scope.

Definition 21 (Architectural TTS) *An Architectural timed transition system $\mathcal{T}$ is a tuple $\langle$ S, ι, Σ, Φ, $\longrightarrow$ $\rangle$ where S is a (possibly infinite) set of states, $\iota \in S$ is the initial state, Σ is a finite set of labels, Φ is a finite set of components and $\longrightarrow \subseteq S \times \Sigma \cup \Phi \cup \mathbb{R}^{\geq 0} \times S$ is the transition relation where $\mathbb{R}^{\geq 0}$ is the set of positive real numbers. If $(q, \sigma, q') \in \longrightarrow$, we write $q \xrightarrow{\sigma} q'$.*

A trajectory of an Architectural TTS $\mathcal{T}= \langle S, \iota, \Sigma, \Phi, \rightarrow\rangle$ is a sequence π = $(s_0,t_0),\ldots,(s_k,t_k)$ such that for $0 \leq i \leq k$, $(s_i,t_i) \in$ S $\times$ $\mathbb{R}$ and for $\leq i < k$, $s_i \xrightarrow{\sigma} s_{i+1}$ and either $\sigma \in \Sigma \cup \Phi$ and $t_{i+1} = t_i$, or $\sigma \in \mathbb{R}^{>0}$ and $t_{i+1} = t_i+\sigma$. A state s of $\mathcal{T}$ is *reachable* if there exists a trajectory π = $(s_0,t_0),\ldots,(s_k,t_k)$ such that $s_0 = \iota$ and $s_n = s$.

Definition 22 (Architectural TA) *An Architectural timed automaton is a tuple $\mathcal{A}$ = $\langle$Loc, C, q_0, Lab, Comp, Edg$\rangle$ where:*

- *Loc is a finite set of locations representing the discrete states of the automaton.*

- *$C = \{ c_1, ..., c_n \}$ is a finite set of real-valued variables.*
- *$q_0 = (l_0, v0)$ where $l_0 \in Loc$ is the initial location and v_0 is the initial clock valuation.*
- *Lab is a finite alphabet of labels.*
- *Comp is a the set of architectural constraints.*
  $Comp \subseteq CompId \times CompType$. Where CompId represents the explicit component Id or "" (to denote the* any component *scope) and CompType = {Process, Agent, Actor, Slot, team, etc.}.*
- *$Edg \subseteq Loc \times Loc \times G \times Lab \times Comp \times 2^C$ is a set of edges. An edge $(l, l', g, \sigma, cp, R)$ represents a jump from location l to location l' with guard* g, *event σ,* component cp *and a subset $R \subseteq$* C *of variables to be reset.*

Architectural constraints can be associated with any qualitative or quantitative temporal logic since no new operators are introduced. In what follows, we present the syntax and the semantics of what we name *Architectural TCTL (ArTCTL).*

9.4.1 Architectural TCTL (ArTCTL)

We extend TCTL logic with an architectural dimension by associating an architectural scope to atomic propositions.

Definition 23 (Syntax of ArTCTL formulas) *Let $\mathcal{A}$ be a timed automaton, $\mathcal{AP}$ a set of atomic propositions,* Comp *a set of architectural constraints (as defined in Definition 22) and D a non-empty set of clocks that is disjoint from the clocks of $\mathcal{A}$, i.e. $C \cap D = \emptyset$. $\sim$ denotes one of the binary relations $<, \leq, =, \geq, >$.*

An ArTCTL formula ϕ has the following syntax rules.

$$\phi ::= p_{cp} \mid \neg\phi \mid \phi \vee \psi \mid z.\phi \mid E[\phi U_{\sim c}\psi] \mid A[\phi U_{\sim c}\psi] \tag{31}$$

where $p \in \mathcal{AP}$, $cp \in$ Comp, *and $z \in D$. z is called the freeze identifier and bounds the clock z in ϕ. For instance, using the freeze identifier the formula $A[\phi_{C1} U_{\leq 5}\ \psi_{C2}]$ can be defined by:*
z in $A[(\phi_{C1} \wedge z \leq 5)\ U\ \psi_{C2}]$

Definition 24 (Semantics of ArTCTL) *The satisfaction relation $(A,s) \models \phi$ (i.e. ϕ is satisfied at state s in TA A) is defined inductively as follows:*

- *$A,s \models p_{cp}$ iff p is true in state s and satisfies constraint* cp *(i.e. let cp = (cpType, cpId), p is true within component cpId of type cpType)*
- *$A,s \models \neg\ \phi$ iff $A,s \not\models \phi$*
- *$A,s \models \phi \vee \psi$ iff either $A,s \models \phi$ or $A,s \models \psi$*
- *$A,s \models z.\phi$ iff $A,s\{z\} \models \phi$*

- *$A,s \models E[\phi\ U_{\sim c}\ \psi]$ iff there exists a run $(s_1, t_1)(s_2, t_2) \ldots$ such that $s_1 = s$ in A and there exist an $i \geq 1$ and a $\delta \in [0, t_{i+1}-t_i]$ such that*
  - *$A, s_i + \delta \models \psi$*
  - *for all j, δ', if either $(1 \leq j < i) \wedge (\delta' \in [0, t_{j+1}-t_j])$ or $(j=i) \wedge (\delta' \in [0,\delta))$, then $A, s_j + \delta' \models \phi$*
- *$A,s \models A[\phi\ U_{\sim c}\ \psi]$ iff for all runs $(s_1, t_1)(s_2, t_2) \ldots$ such that $s_1 = s$ in A and there exist an $i \geq 1$ and a $\delta \in [0, t_{i+1}-t_i]$ such that*
  - *$A, s_i + \delta \models \psi$*
  - *for all j, δ', if either $(1 \leq j < i) \wedge (\delta' \in [0, t_{j+1}-t_j])$ or $(j=i) \wedge (\delta' \in [0,\delta))$, then $A, s_j + \delta' \models \phi$*

For instance, the absence property "*P does never occur within the component CpId of type CpType*" can be expressed in ArTCTL with: $AG(\neg P_{(CpId,CpType)})$.

9.5 Applying Property Patterns

In this section, we apply our pattern system to the case study of *IP Header Compression* feature. This example is more suitable to illustrate our property pattern approach than the running case study of the telephone system presented in Section 2.1.4.

9.5.1 Case Study: IP Header Compression Feature

As networks evolve to provide more bandwidth, the applications, services and the consumers of those applications are competing for that bandwidth. In many services and applications, such as voice and video over IP, several fields in the header of a given flow remain constant for the length of the flow. IP header compression (IPHC) achieves major gain in terms of packet compression because although some fields in the header change in every packet, the difference from packet to packet is often constant, and therefore the second-order difference is zero. The decompressor can reconstruct the original header without any loss of information. IPHC is a hop-by-hop compression scheme (i.e., works on a point-to-point link). IP header compression can improve throughput and reduce packet loss and delay.

IPHC Preconditions

Before any IP packets may be communicated, *PPP* (which allows two machines on a point-to-point communication link to negotiate various parameters for authentication) and *IPCP* (responsible for configuring, enabling, and disabling the IP protocol modules on both ends of the point-to-point link) negotiations must be completed successfully. Figure 9.14 describes this negotiation phase using the shared responsibility construct.

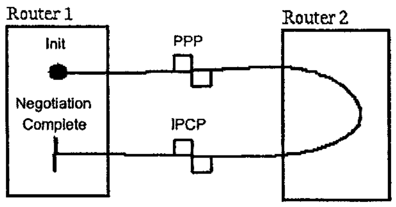


Figure 9.14: PPP and IPCP Negotiation

Compression/Decompression Types

Mainly three types of compression were presented in RFC 2507 [DNP99], RFC2508 [CJ99] and RFC1144 [Jac90]: RTP (RTP compression: Real Time Protocol), cUDP (UDP compression) and cTCP (TCP compression). For nonRTP traffic another type of compression called nonTCP can be used as well.

Compression/decompression takes place either in the fast Path (ASIC forwarding) or the slow path (software forwarding) depending on the type of traffic. A possible design of IPHC may consider compressing/decompressing cRTP and cUDP traffic on a fast path while compressing/decompressing cTCP traffic on slow path since protocol packets over the TCP transport would constitute a significantly lower percentage of all traffic in typical application profiles that use IPHC.

IP Header Compression Requirements

Compression Scenario

Figure 9.15 illustrates a high level view of the compression scenario. UCM start point *Rec-Uncompr-Packet* denotes the reception of a non compressed packet. Then the router checks whether the egress interface (towards the destination) is IPHC enabled. If the egress interface is not IPHC-enabled, the packet is then forwarded uncompressed towards its destination (i.e. Dynamic responsibility *Send-Uncompressed*). Otherwise, the compressor checks the packet type to distinguish compressible packets. The design presented in Figure 9.15 does not consider plain IP packets and IP packets with options for compression.

Compressible packets (RTP, UDP and TCP) are looked up in a repository of packet headers (i.e., UCM responsibility *HeaderLookup*). If a matching header (that corresponds to the context of a given flow) is found, the incoming packet is compressed. If no match is found, the packet header is copied into that repository and a new context is defined. Then depending on the protocol type the corresponding compression type is selected and applied to the packet (i.e., cRTP, cUDP or cTCP). Compression latency is expected to be within 100μsec.

Decompression Scenario

Figure 9.16 illustrates a high level view of the decompression scenario. The decompressor should handle two types of packets: (a) Full Header and (b) Compressed packet. The start of a compressed

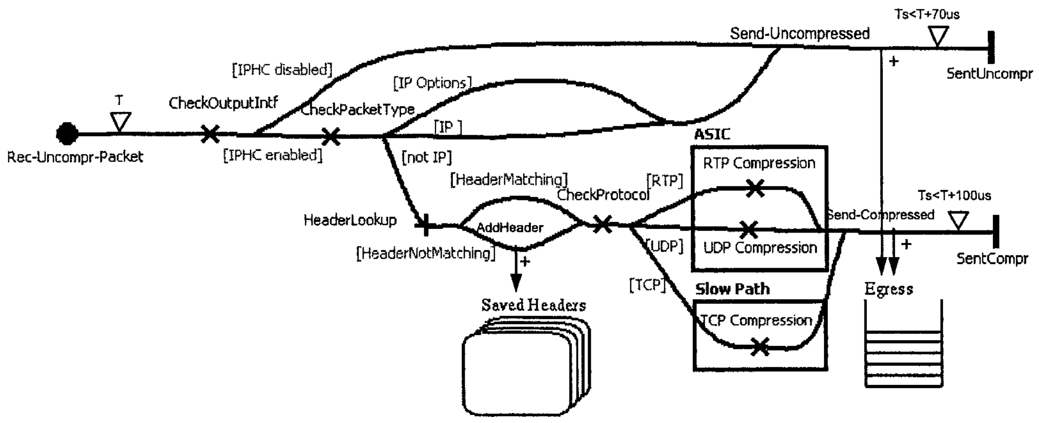


Figure 9.15: IPHC: Compression Scenario

flow is indicated by the arrival of a Full header. The decompressor will store the contents of the header from the Full header packet (i.e., responsibility *StoreContextID*). Subsequent compressed packets will be decompressed by using the stored context from the Full header packet (i.e., *RetrieveContextID*) and the information present in the compressed packet. If there is no matching with the stored context ID or the packet is out of sequence, then the packet is discarded and a context state packet (CS packet) is sent to the compressor to notify that something wrong happened (i.e., *GenerateCS-Packet*).

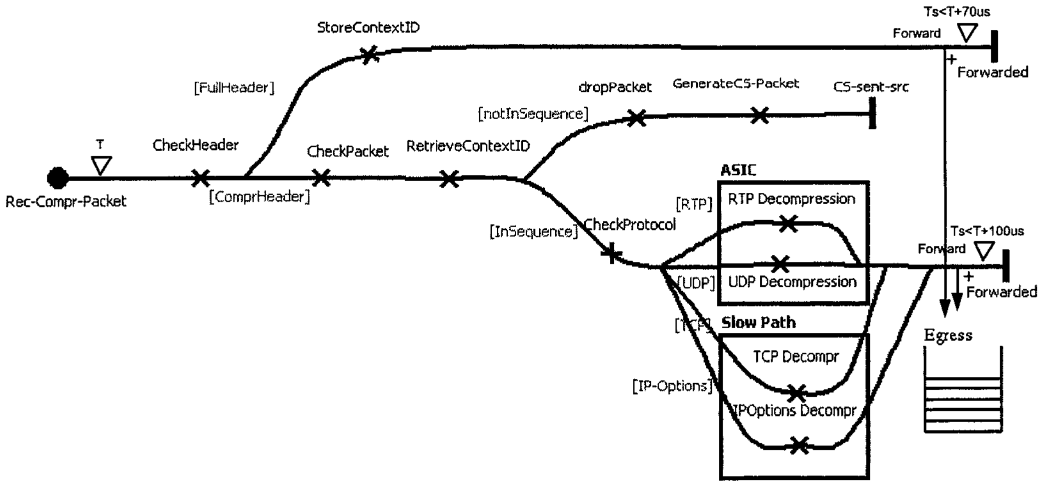


Figure 9.16: IPHC: Decompression Scenario

9.5.2 IPHC System Properties

- **Requirement 1: An RTP packet is compressed in the fast path (ASIC) and the latency is less than 150μsec.**

  This requirement is described in Figure 9.17. Its corresponding architectural TCTL formula is:

  $$\neg E[\neg RTPpacket\ U\ (RTPpacket_{(Team,ASIC)} \wedge EG_{\leq 150} \neg RTPcompr)] \qquad (32)$$

  The design shown in Figure 9.15 satisfies this property since RTP flows are compressed in the fast path (ASIC) and the latency is within the acceptable range (100μsec<150μsec).

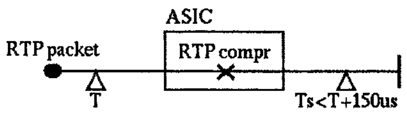


Figure 9.17: Bounded Existence Property Satisfying IPHC Design

- **Requirement 2: A TCP packet is compressed in the fast path (ASIC) and the latency is less than 50μsec.**

  This requirement is described in Figure 9.18. Its corresponding ArTCTL formula is:

  $$\neg E[\neg TCPpacket\ U\ (TCPpacket_{(Team,ASIC)} \wedge EG_{\leq 50} \neg TCPcompr)] \qquad (33)$$

  In this case the design in Figure 9.15 violates this property since TCP flow compression takes place in slow path (software) and the latency is lower (50μsec<100μsec) than the one specified in the property.

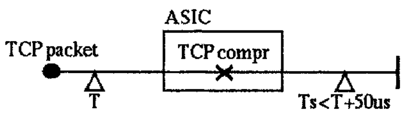


Figure 9.18: Bounded Existence Property Violating IPHC Design

- **Requirement 3: In the compression scenario, the header lookup is followed by a protocol check.**

  This requirement is described in Figure 9.19. Its corresponding CTL formula is:

  $$AG(HeaderLookup \Rightarrow AF(CheckProtocol)) \qquad (34)$$

  The design in Figure 9.15 satisfies this property since responsibility *HeaderLookup* is followed by responsibility *CheckProtocol* for all paths that contain *HeaderLookup*.

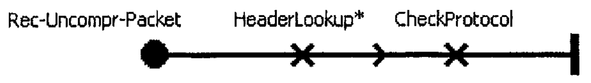


Figure 9.19: Response Property Satisfying IPHC Design

- Requirement 4: **IP options packets are not compressed.**
  This requirement is described in Figure 9.20. Its corresponding architectural CTL formula is:

$$AG(\neg IPOptionsCompression_{(Team,*)}) \quad (35)$$

The design in Figure 9.15 satisfies this property since IP options packets are not compressed.

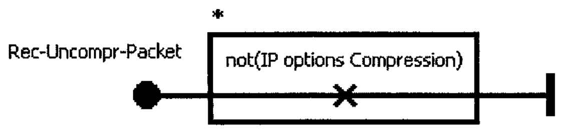


Figure 9.20: Absence Property Satisfying IPHC Design

- Requirement 5: **In the decompression scenario, packet drop is always preceded by storing the context ID**
  This requirement is described in Figure 9.21. Its corresponding CTL formula is:

$$\neg E[\neg StoreContextID\ U(dropPacket \wedge negStoreContextID)] \quad (36)$$

This property is not satisfied since responsibilities *StoreContextID* and *dropPacket* belong to two distinct paths in Figure 9.16.

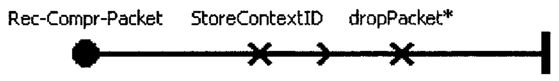


Figure 9.21: Precedence Property Violating IPHC Design

9.6 Chapter Summary

Specification building is one of the most difficult activities of model-based verification. The work presented in this chapter has yielded two main contributions. First, we have presented a UCM based specification pattern that can simplify this activity and make it available to the novice practitioner.

The specification pattern system uses templates to cover most common expected properties found in requirements specifications. We have provided a mapping of our UCM-based system to popular temporal logics CTL and TCTL. These templates combine qualitative, real-time and architectural properties into a single graphical representation. To the best of our knowledge, no existing pattern system has considered these three scopes together. However, we do not claim that our real-time specification pattern system is complete. Second, we have extended the traditional real-time temporal logics to include architectural aspects. We have given an overview of the semantics of the systems targeted by what we call "*Architectural real-time temporal logic*". Moreover, we have provided formal syntax and semantics of *ArTCTL*, an extension of TCTL with architectural aspects. We believe that having the requirement specification and properties described using the same formalism will enable greater degrees of analysis while preserving a high level of abstraction.

Chapter 10

Experiments with Early Stages Validation and Verification Methodology

In the previous chapters, we have explained how the *Early Stages V&V Methodology* can be applied to validate and verify UCM specifications. We have used the simple telephone system as a running example to illustrate the different steps of our proposed methodology.

Use Case Maps have been used to model the dynamics of complex systems in domains as telecommunications and e-business process modeling. In this chapter, we show that our proposed methodology can be applied to more complex applications and various domains. We present two case studies:

- Real-time Systems: We will investigate the use of our proposed approach to describe, validate and verify the IP Multicast routing protocol[1] (see Section 10.1).
- Process Business Models: we will investigate the use of our proposed approach to validate and verify an existing online store business model that was introduced in [ARW05] (see Section 10.2).

10.1 Case Study 1: IP Multicast Routing Protocol

10.1.1 Introduction to IP Multicast

IP Multicast is a technique for many-to-many communication over an IP infrastructure. Typical Multicast applications include VAT (Visual Audio Tool), VIC (Video conferencing) and WB (white board). Figure 10.1 [Inc04], illustrates a network topology with two sources (source 1 and source 2), two receivers (receiver 1 and receiver 2) and six multicast routers (A, B, C, D, E and F). This

[1]One scenario of this case study is published in SDL Forum - SDL 2007) [HRD07a]

topology will be used throughout the cast study to illustrate different scenarios. To send data to multiple destinations using unicast, the sender has to send for each receiver its own data flow. The sender has to make copies of the same packet and send them once for each receiver. In multicast, the sender sends only one copy of a single data packet addressed to a group of receivers - multicast group. Multicast transmission provides many advantages over unicast transmission in one-to-many or many-to-many environment:

- Enhanced efficiency: Available network bandwidth is utilized more efficiently because multiple streams of data are replaced with a single transmission.
- Optimized Performance: Fewer copies of data require forwarding and processing.
- Distributed Applications: Multipoint applications will not be possible as demand and usage grows, because unicast transmission will not scale.

Multicast routers replicate and forward the packet to all the branches where receivers may exist. Receivers express their interest in multicast traffic by registering at their first-hop router using the Internet Group Management Protocol (IGMP)for IPv4 hosts or Multicast Listener Discovery (MLD) for IPv6 hosts (Receivers 1 and 2 register respectively with routers C and E and Last-hop routers (leafs: C and E) communicate group membership to the network). The multicast network routers learn about their multicast-enabled neighbors to build appropriate distribution tree (Distribution trees are shown with arrows in Figure 10.1) to prevent loops and to apply scoping and filtering. Inside the multicast network, there are various multicast routing protocols used. They may be separated into these two groups: Intra-domain (for example PIM, Distance Vector Multicast Routing Protocol (DVMRP), Multicast Open Shortest-Path First (MOSPF), Core Bases Trees (CBTs)) and inter-domain (for example, Multi-protocol BGP Extension for IP Multicast [MBGP]).

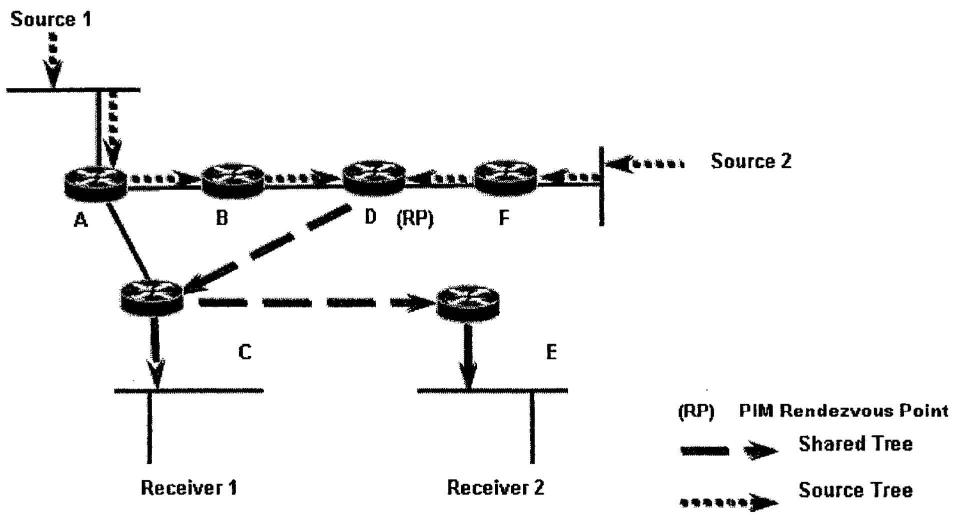


Figure 10.1: Example of a Multicast Enabled Network [Inc04]

The multicast addresses are within the class D and are in the range 224.0.0.0 through 239.255.255.255. The range of addresses between 224.0.0.0 and 224.0.0.255, inclusive, is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting. Multicast routers should not forward any multicast datagram with destination addresses in this range, regardless of its TTL (Time to Live). In this case study, we treat only IPv4 Multicast and we focus only on PIM protocol.

10.1.2 IP Multicast: UCM Scenarios Specifications

In this section, we present IP multicast scenarios and their corresponding UCMs.

- **Multicast Forwarding Scenario:**
  In unicast routing, when a router receives a packet, the decision about where the packet should be forwarded to depends on the destination address of the packet itself. In multicast routing, the decision on where a particular packet should be forwarded to depends on the origin of the packet.

  - **Reverse Path Forwarding (RPF):** Multicast routing uses a mechanism called Reverse Path Forwarding (RPF) to prevent forwarding loops and to ensure the shortest path from the source to the receivers. Routers perform an RPF check, by examining the unicast routing table, to ensure that arriving multicast packets were received through the interface that is on the most direct path to the source that sent the packets. If the RPF check succeeds, the datagram is forwarded on each interface in the outgoing interface list (the packet is never sent back out of the RPF interface). If RPF check fails, the datagram is typically silently discarded. This scenario is illustrated in Figure 10.2.

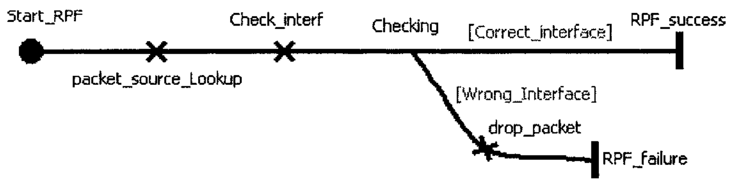


Figure 10.2: RPF Check Plug-in Map

The checking of the incoming interface may result in a wrong interface (i.e., guard wrong_interface) that leads to the packet drop (i.e., responsibility drop_packet) and RFP failure (i.e., end point RPF_failure). A packet arriving at the right interface leads to a successful RPF check. RPF_check scenario represents a plug-in of a larger collection of scenarios (i.e., Multicast Forwarding), whose root map is shown in Figure 10.3. Start point *Start_RPF* is bounded to *IN1*, end point *RPF_success* is bounded to *OUT1* and end point *RPF_failure* is bounded to *OUT2*.

  - **TTL Threshold:** In order to keep some external multicast traffic out of their network,

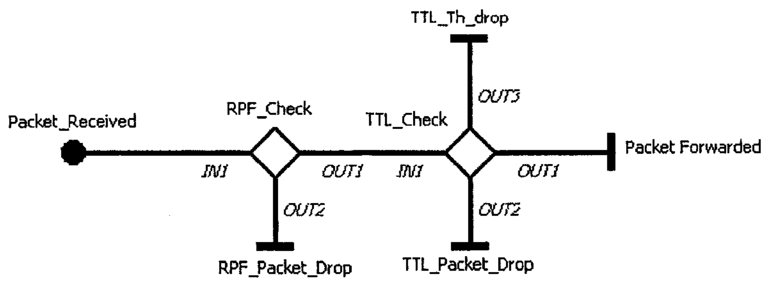


Figure 10.3: Multicast Forwarding

service providers use a *TTL threshold* mechanism to define boundaries for certain multicast traffic. All incoming IP packets first have their TTL decremented by one. If the resulting TTL is less than or zero, it is dropped. If a multicast packet is to be forwarded out of an interface with a non-zero TTL threshold, then its TTL is checked against the TTL threshold. If a packet TTL is less than the specified threshold, it is not forwarded out of the interface. Figure 10.4 shows the corresponding UCM plug-in.

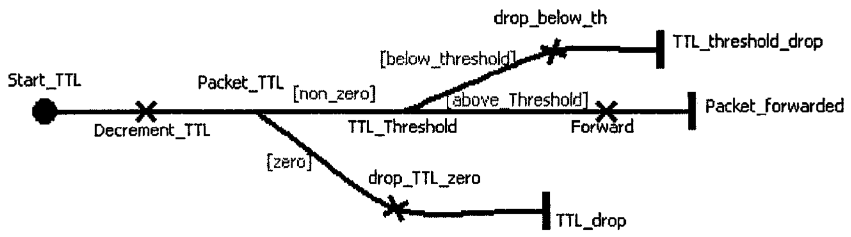


Figure 10.4: TTL Check Plug-in Map

- **Reporting Group Membership:** The primary purpose of IGMP is to permit hosts to communicate their desire to receive multicast traffic to the IP multicast router(s) on the local network. This action, in turn, permits the IP multicast router(s) to *join* the specified multicast group and to begin forwarding the multicast traffic onto the network segment. The scenarios given below are specific to IGMPv2 defined in RFC2236 [Fen97].
  - **Group-specific queries:** A group-specific query allows the router to query memberships only in a single group instead of all groups, illustrated by Figure 10.5. In this example it is assumed that Receiver 1 and Receiver 2 are part of one multi-access connection and that last_hop_Router denotes the leaf router connected to this multi-access connection (e.g., Ethernet connection). The last-hop router sends a single query group message (i.e., responsibility Query_group) to receiver 1 and receiver 2. When a receiver receives the query and he is not a member of the queried group, he ignores the query. Otherwise, the receiver starts a randomized count down timer. When the timer reaches zero, the receiver

sends a *Membership Report* to notify the router that the group is still active. However if a host receives a Membership Report (from another receiver) before its associated count-down timer reaches zero, it cancels the count-down timer thereby suppressing its own report. These two concurrent scenarios (located in receivers 1 and 2) are illustrated in Figure 10.5 by using the AND-Fork construct.

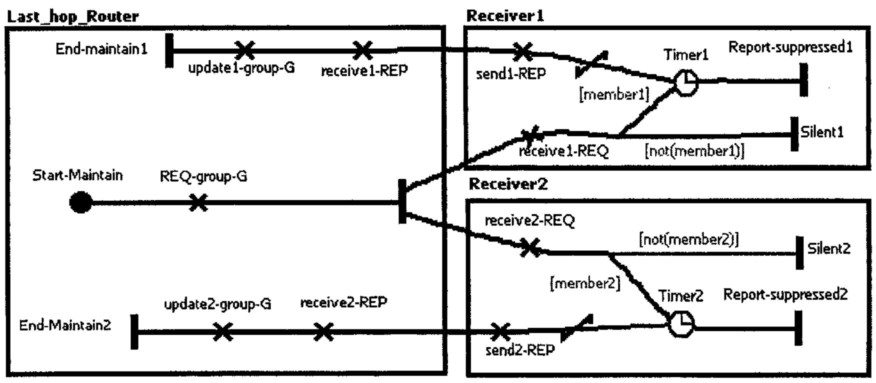


Figure 10.5: IP Multicast Group Membership Maintenance

– **Joining a multicast group:** Receivers joining a multicast group do not have to wait for a query and send an unsolicited report indicating their interest in joining a multicast group. This scenario is illustrated in Figure 10.6.

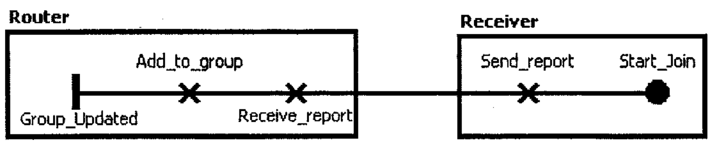


Figure 10.6: Join IP Multicast Group

– **Leaving a multicast group:** IGMPv2 leave group message allows hosts to tell the local multicast router that they are leaving the group. This scenario is illustrated in Figure 10.7. Upon receipt of the Leave Group message, the router sends out a group-specific query (see Figure 10.5) and determines whether there are any remaining hosts interested in receiving the traffic. If there are no replies, the router times out the group and stops forwarding the traffic.

- **Protocol Independent Multicast (PIM):** PIM-SM is defined by the IETF [2] in RFC2362 [EFH+98]. Protocol Independent Multicast (PIM) has two modes Sparse Mode and Dense Mode. In this case study, we focus on the Sparse Mode (PIM-SM) model. PIM-SM is independent of underlying unicast protocols. PIM uses a rendezvous point (RP) to coordinate

[2]http://www.ietf.org

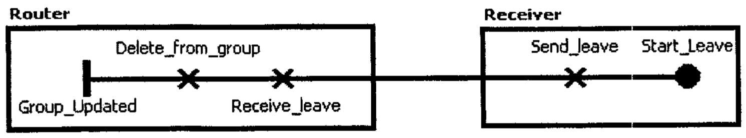


Figure 10.7: Leave IP Multicast Group

forwarding of multicast traffic from a source to receivers. Senders register with the RP and send a single copy of multicast data through it to the registered receivers. Group members are joined to the shared tree by their local designated router (DR). A shared tree is built which is rooted at the RP.

- **PIM-SM Shared Tree Join:** The last-hop router knows the IP address of the RP router for a group G, and it sends a join for this group toward the RP. The join travels hop-by-hop toward the RP building a branch of the shared tree (by adding (*, G) to multicast table, '*' denotes any source) that extends from the RP to the last-hop router. At this point, group G traffic may flow down the shared tree to the receiver. In the example given in Figure 10.1, the router E sends a join to router D which is the defined RP. The join travels through router C resulting in the construction of the shared tree.

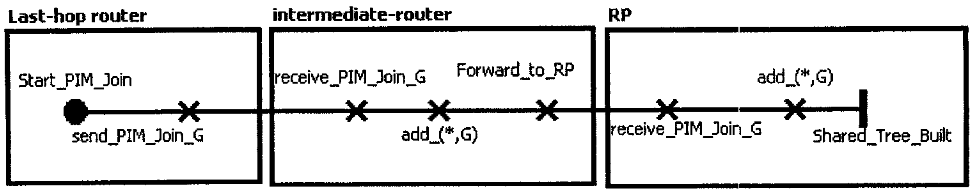


Figure 10.8: PIM-SM Join

- **PIM-SM Sender Registration:** In the case of an active source for group G starts sending multicast traffic, its designated router (DR) registers this source with the RP. In the example given in Figure 10.1, Routers A and F, which are respectively the designated routers for source 1 and source 2, register to router D (defined RP). This process results in the construction of the source tree. To register the source, the DR encapsulates the multicast packets in a *Register* message and unicast it to the RP. When the RP receives the *Register* message, it decapsulates the multicast packets and starts sending them down the shared tree towards the receivers. At the same time, the RP starts building a shortest-path tree (SPT) by sending joins towards the source. After the SPT is built from the designated router to the RP, the multicast traffic starts to flow from the source to the RP without being encapsulated in Register messages. RP sends a *Register Stop* message to the source's designated router to inform it may stop sending unicast Register messages.

  PIM-SM has the capability for the last-hop router to switch to the shortest-path tree

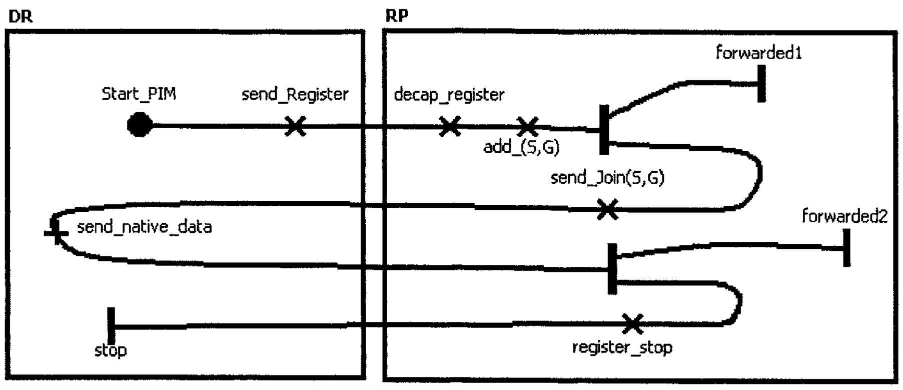


Figure 10.9: PIM-SM Sender Registration

(SPT) and bypass the RP, if the traffic rate is above a set threshold called the SPT threshold. The resulting scenarios are not described in this thesis.

10.1.3 AsmL Specification of IP Multicast Protocol

Figure 10.10 illustrates the AsmL specification of the Multicast Forwarding scenario introduced in Figure 10.3.

```
//Global Variables
var TTL_zero = new BooleanValue(false)
var TTL_below_threshold = new BooleanValue(true)
var RPF_correct_interface = new BooleanValue(true)
```

```
// Stub RPF : RPF Check Plug-in
var Start_RPF as SP_Construct = SP_Construct (RPF_in1, r1 , "Start_RPF", BooleanVar(pre_cond_start),Router)
var packet_source_lookup as R_Construct = R_Construct(r1, r2, "packet_source_lookup", Router)
var check_interf as R_Construct = R_Construct(r2, r3, "check_interf", Router)
var drop_packet as R_Construct = R_Construct(r5, r6, "drop_packet", Router)
var RPF_success as EP_Construct = EP_Construct(r4 , h0 , "RPF_success", true, Router)
var RPF_failure as EP_Construct = EP_Construct(r6 , h0 , "RPF_failure", true, Router)
var RFP_OF as OF_Construct = OF_Construct(r3 , {OR_Selection(r4, BooleanVar(RPF_correct_interface)) ,
OR_Selection(r5, -BooleanVar(RPF_correct_interface))}, "RFP_OF", Router)
var RPF_check_plugin as Maps = Maps ("RPF_check_plugin", {UCMElement(Start_RPF, r1,
packet_source_lookup), UCMElement(packet_source_lookup, r2, check_interf), UCMElement(check_interf, r3,
RFP_OF), UCMElement(RFP_OF, r4, RPF_success), UCMElement(RPF_success, h0, RPF_success), UCMEle-
ment(RFP_OF, r5, drop_packet), UCMElement(drop_packet, r6, RPF_failure), UCMElement(RPF_failure, h0 ,
RPF_failure)}, {RPF_success, RPF_failure})
var SRPF_Check as Stub_Construct = Stub_Construct(e1,e2,e3,{Stub_Selection (RPF_check_plugin, Boolean-
Var(_true))}, {Stub_Binding(RPF_check_plugin, e1, Start_RPF), Stub_Binding (RPF_check_plugin, e2,
RPF_failure), Stub_Binding (RPF_check_plugin, e3, RPF_success)}, "SRPF_Check")
```

```
// TTL Check plug-in map
var Start_TTL as SP_Construct = SP_Construct (TTL_in1 , t1, " Start_TTL", BooleanVar(pre_cond_start),
Router)
var Decrement_TTL as R_Construct = R_Construct(t1 , t2, " Decrement_TTL", Router)
var TTL_zero_OF as OF_Construct = OF_Construct(t2 , {OR_Selection(t3, -BooleanVar(TTL_zero)) ,
OR_Selection(t9, BooleanVar(TTL_zero))}, " TTL_zero_OF ", Router)
var TTL_Threshold_OF as OF_Construct = OF_Construct(t3, {OR_Selection(t4, Boolean-
Var(TTL_below_threshold)), OR_Selection(t6, -BooleanVar(TTL_below_threshold))}, "TTL_Threshold_OF",
Router)
var drop_TTL_zero as R_Construct = R_Construct(t9, t10,"drop_TTL_zero", Router)
var drop_below_threshold as R_Construct = R_Construct(t4, t5," drop_below_threshold ", Router)
var Forward as R_Construct = R_Construct(t6, t8," Forward ", Router)
var TTL_threshold_drop as EP_Construct = EP_Construct(t5 , h0 , "TTL_threshold_drop ", true, Router)
var Packet_forwarded as EP_Construct = EP_Construct(t8 , h0 , "Packet_forwarded ", true, Router)
var TTL_drop as EP_Construct = EP_Construct(t10 , h0 , "TTL_drop", true, Router)
var TTL_plugin as Maps = Maps("TTL_plugin ",{UCMElement(Start_TTL, t1, Decrement_TTL),
UCMElement(Decrement_TTL, t2, TTL_zero_OF), UCMElement(TTL_zero_OF, t3, TTL_Threshold_OF),
UCMElement(TTL_zero_OF, t9, drop_TTL_zero), UCMElement(drop_TTL_zero, t10, TTL_drop),
UCMElement(TTL_Threshold_OF, t4, drop_below_threshold), UCMElement(drop_below_threshold,
t5, TTL_threshold_drop), UCMElement(TTL_Threshold_OF, t6, Forward), UCMElement(Forward,
t8 , Packet_forwarded), UCMElement(TTL_threshold_drop, h0 , TTL_threshold_drop), UCMEle-
ment(Packet_forwarded, h0 , Packet_forwarded), UCMElement(TTL_drop, h0 , TTL_drop)}, {TTL_threshold_drop,
Packet_forwarded, TTL_drop})
var STTL_Check as Stub_Construct= Stub_Construct({e3},{e4,e5,e6},{Stub_Selection (TTL_plugin,
BooleanVar(_true))}, {Stub_Binding(TTL_plugin, e3, Start_TTL), Stub_Binding (TTL_plugin, e4,
Packet_forwarded),Stub_Binding (TTL_plugin,e5, TTL_threshold_drop), Stub_Binding (TTL_plugin, e6,
TTL_drop)}, "STTL_Check")
```

```
// Root map constructs
var Packet_Received as SP_Construct = SP_Construct (in1 , e1 , "Packet Received", BooleanVar(pre_cond_start),
Router)
var RPF_Packet_Drop as EP_Construct = EP_Construct(e2 , h0 , "RPF_Packet_Drop", true, Router)
var Packet_Forwarded_root as EP_Construct = EP_Construct(e4 , h0 , " Packet_Forwarded_root", true, Router)
var TTL_th_drop as EP_Construct = EP_Construct(e5 , h0 , "TTL_th_drop", true, Router)
var TTL_Packet_Drop as EP_Construct = EP_Construct(e6 , h0 , "TTL_Packet_Drop", true, Router)
```

```
var RootMap as Maps = Maps("RootMap", {UCMElement(Packet_Received, e1, SRPF_Check), UCMEle-
ment(SRPF_Check, e2 , RPF_Packet_Drop), UCMElement(RPF_Packet_Drop, h0, RPF_Packet_Drop),
UCMElement(SRPF_Check, e3, STTL_Check), UCMElement(STTL_Check, e4, Packet_Forwarded_root),
UCMElement(STTL_Check, e5, TTL_th_drop), UCMElement(STTL_Check, e6, TTL_Packet_Drop), UCMEle-
ment(TTL_Packet_Drop, h0, TTL_Packet_Drop), UCMElement(TTL_th_drop, h0, TTL_th_drop), UCMEle-
ment(Packet_Forwarded_root, h0, Packet_Forwarded_root)}, { Packet_Forwarded_root, TTL_th_drop,
TTL_Packet_Drop})
```

Figure 10.10: AsmL Implementation of the Multicast Forwarding Scenario

Figure 10.11 illustrates the *AsmL* specification of the Group Maintenance scenario described in Figure 10.5.

```
//Global Variables
var Rec1_member = new BooleanValue(true)
var Rec2_member = new BooleanValue(false)
var Report_sent = new BooleanValue(false)

// Root map constructs:
var start_maintain as SP_Construct = SP_Construct (in1, m1 ,"start_maintain", BooleanVar(pre_cond_start),
Last_hop_Router)
var REQ_group_G as R_Construct = R_Construct(m1, m2," REQ_group_G ", Last_hop_Router)
var Maintain_AF as AF_Construct = AF_Construct(m2 ,m3, m12, "Maintain_AF", Last_hop_Router)
var Receive1_REQ as R_Construct = R_Construct(m3, m4," Receive1_REQ ", Receiver1)
var Receiver1_member_OF as OF_Construct = OF_Construct(m4 , {OR_Selection(m5, BooleanVar(Rec1_member))
, OR_Selection(m6, -BooleanVar(Rec1_member))}, "Receiver1_member ", Receiver1)
var Timer1 as Timer_Construct = Timer_Construct(m5 , {OR_Selection(m7, BooleanVar(Report_sent)) ,
OR_Selection(m8, -BooleanVar(Report_sent))}, "Receiver1_member", Receiver1)
var Send1_REP as R_Construct = R_Construct(m8, m9," Send1_REP", Receiver1)
var Silent1 as EP_Construct = EP_Construct(m6 , h0 , " Silent1 ", true, Receiver1)
var Report_suppressed1 as EP_Construct = EP_Construct(m7, h0 , "Report_suppressed1", true, Receiver1)
var Receive1_REP as R_Construct = R_Construct(m9, m10," Receive1_REP", Last_hop_Router)
var Update1_group_G as R_Construct = R_Construct(m10, m11,"Update1_group_G", Last_hop_Router)
var End_maintain1 as EP_Construct = EP_Construct(m11 , h0 , "End_maintain1", true, Last_hop_Router)
var Receive2_REQ as R_Construct = R_Construct(m12, m13," Receive2_REQ ", Receiver2)
var Receiver2_member_OF as OF_Construct = OF_Construct(m13 , {OR_Selection(m15, Boolean-
Var(Rec2_member)) , OR_Selection(m14, -BooleanVar(Rec2_member))}, "Receiver2_member ", Receiver2)
var Timer2 as Timer_Construct = Timer_Construct(m15 , {OR_Selection(m16, BooleanVar(Report_sent)) ,
OR_Selection(m17, -BooleanVar(Report_sent))}, "Receiver2_member", Receiver2)
var Send2_REP as R_Construct = R_Construct(m17, m18," Send2_REP", Receiver2)
var Silent2 as EP_Construct = EP_Construct(m14 , h0 , " Silent2 ", true, Receiver2)
var Report_suppressed2 as EP_Construct = EP_Construct(m16, h0 , "Report_suppressed2", true, Receiver2)
var Receive2_REP as R_Construct = R_Construct(m18, m19," Receive2_REP", Last_hop_Router)
var Update2_group_G as R_Construct = R_Construct(m19, m20,"Update2_group_G", Last_hop_Router)
var End_maintain2 as EP_Construct = EP_Construct(m20 , h0 , "End_maintain2", true, Last_hop_Router)

var RootMap as Maps = Maps("RootMap", {UCMElement(start_maintain, m1, REQ_group_G), UCMEle-
ment(REQ_group_G, m2, Maintain_AF), UCMElement(Maintain_AF,m3 , Receive1_REQ), UCMEle-
ment(Maintain_AF, m12 , Receive2_REQ), UCMElement(Receive1_REQ, m4, Receiver1_member_OF),
UCMElement(Receiver1_member_OF, m6, Silent1), UCMElement(Receiver1_member_OF, m5, Timer1),
UCMElement(Timer1, m7 , Report_suppressed2), UCMElement(Timer1, m8 , Send1_REP), UCMEle-
ment(Send1_REP, m9, Receive1_REP), UCMElement(Receive1_REP, m10 , Update1_group_G), UCMEle-
ment(Update1_group_G, m11, End_maintain1), UCMElement(Receive2_REQ, m13, Receiver2_member_OF),
UCMElement(Receiver2_member_OF, m14, Silent1), UCMElement(Receiver2_member_OF, m15, Timer2),
UCMElement(Timer2, m16 , Report_suppressed2), UCMElement(Timer2, m17 , Send2_REP), UCMEle-
ment(Send2_REP, m18, Receive2_REP), UCMElement(Receive2_REP, m19 , Update2_group_G), UCMEle-
ment(Update2_group_G, m20, End_maintain2), UCMElement(End_maintain2, h0, End_maintain2), UCMEle-
ment(End_maintain1, h0, End_maintain1), UCMElement(Report_suppressed1, h0, Report_suppressed1), UCMEle-
ment(Report_suppressed2, h0, Report_suppressed2), UCMElement(Silent1, h0, Silent1), UCMElement(Silent2, h0,
Silent2)}, {End_maintain1, End_maintain2, Silent1, Silent2, Report_suppressed1, Report_suppressed2})
```

Figure 10.11: AsmL Implementation of the Group Maintenance Scenario

Figure 10.12 illustrates the *AsmL* specification of the PIM-SM Sender Registration scenario introduced in Figure 10.9.

```
var start_PIM as SP_Construct = SP_Construct (in1, p1 ,"start_PIM", BooleanVar(pre_cond_start), DR)
var send_Register as R_Construct = R_Construct(p1, p2," send_Register ", DR)
var decap_Register as R_Construct = R_Construct(p2, p3," decap_Register", RP)
var add_S_G as R_Construct = R_Construct(p3, p4,"add_S_G", RP)
var PIM_AF1 as AF_Construct = AF_Construct(p4 ,{p5, p6}, " PIM_AF1 ", RP)
var send_join_S_G as R_Construct = R_Construct(p6, p7," send_join_S_G ", RP)
var send_native_data as R_Construct = R_Construct(p7, p8," send_native_data ", DR)
var PIM_AF2 as AF_Construct = AF_Construct(p8 ,{p9, p10}, " PIM_AF2 ", RP)
var register_stop as R_Construct = R_Construct(p10, p11," register_stop ", RP)
var forwarded1 as EP_Construct = EP_Construct(p5 , h0 , " forwarded1 ", true, RP)
var forwarded2 as EP_Construct = EP_Construct(p9 , h0 , " forwarded2", true, RP)
var stop as EP_Construct = EP_Construct(p11 , h0 , "stop ", true, DR)
```

```
var RootMap as Maps = Maps("RootMap", { UCMElement(start_PIM, p1, send_Register), UCMEle-
ment(send_Register, p2, decap_Register), UCMElement(decap_Register, p3, add_S_G), UCMElement(add_S_G,
p4, PIM_AF1), UCMElement(PIM_AF1, p5, forwarded1), UCMElement(forwarded1, h0, forwarded1),
UCMElement(PIM_AF1, p6, send_join_S_G), UCMElement(send_join_S_G, p7 , send_native_data), UCMEle-
ment(send_native_data, p8 , PIM_AF2), UCMElement(PIM_AF2, p9, forwarded2), UCMElement(PIM_AF2, p10
, register_stop), UCMElement(register_stop, p11, stop), UCMElement(stop, h0, stop), UCMElement(forwarded2,
h0, forwarded2)},{forwarded1, forwarded2, stop})
```

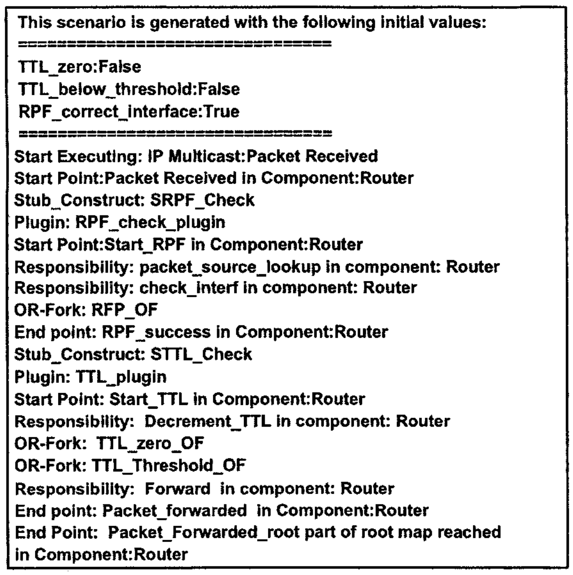
Figure 10.12: AsmL Implementation of PIM-SM Sender Registration

10.1.4 IP Multicast Protocol Scenarios Generation

Figure 10.13 illustrates four traces of the IP multicast forwarding scenario described in Figure 10.3.

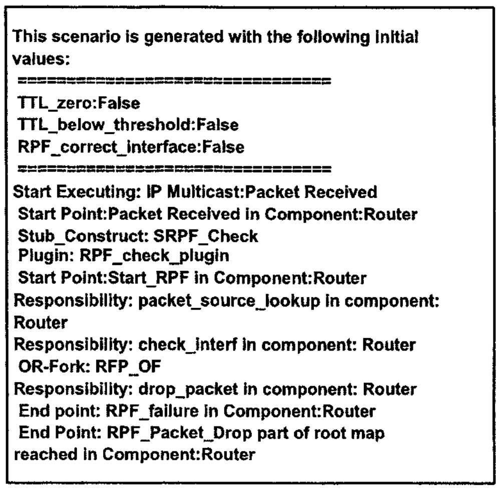
Figure 10.14 illustrates four traces of the IP Multicast Group Membership Maintenance described in Figure 10.5. A simple inspection of the fourth trace reveals that both receivers have sent reports. Hence, the group membership is updated twice. This specification flaw is due to the fact that the timer timeout event and responsibility *SEND_REP* occur in two distinct steps. The non atomic execution of these two events resulted in an interleaving of *Send1_REP* and *Send2_REP*. In the following section, we will show how this flaw is detected using model checking.

Figure 10.15 illustrates the execution trace of the PIM-SM sender registration scenario described in Figure 10.9.



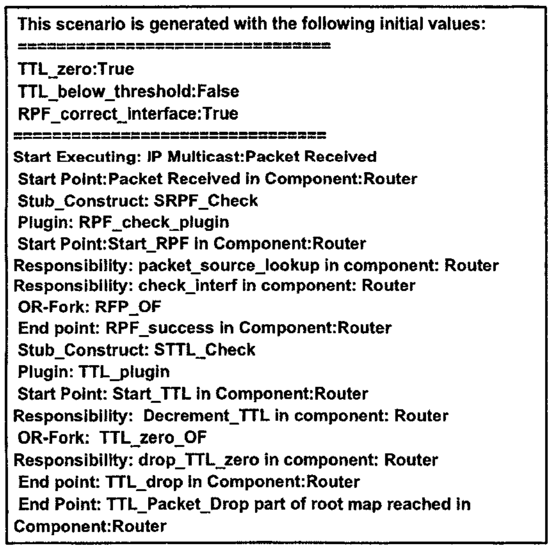
```
This scenario is generated with the following initial values:
================================
TTL_zero:False
TTL_below_threshold:False
RPF_correct_interface:True
================================
Start Executing: IP Multicast:Packet Received
Start Point:Packet Received in Component:Router
Stub_Construct: SRPF_Check
Plugin: RPF_check_plugin
Start Point:Start_RPF in Component:Router
Responsibility: packet_source_lookup in component: Router
Responsibility: check_interf in component: Router
OR-Fork: RFP_OF
End point: RPF_success in Component:Router
Stub_Construct: STTL_Check
Plugin: TTL_plugin
Start Point: Start_TTL in Component:Router
Responsibility:  Decrement_TTL in component: Router
OR-Fork:  TTL_zero_OF
OR-Fork: TTL_Threshold_OF
Responsibility:  Forward  in component: Router
End point: Packet_forwarded  in Component:Router
End Point:  Packet_Forwarded_root part of root map reached
in Component:Router
```

(a) Multicast Forwarding Trace 1



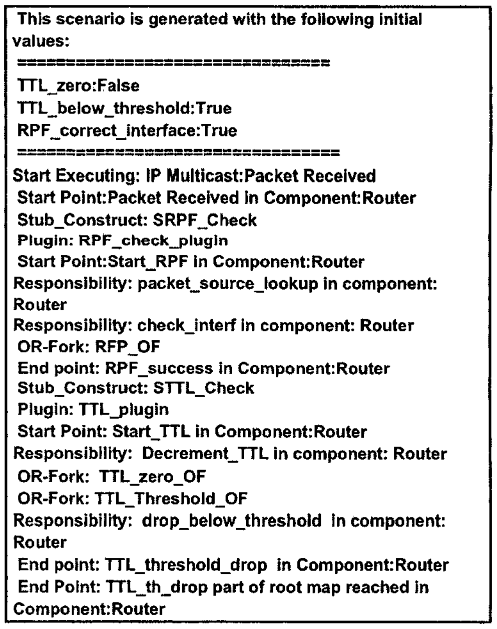
```
This scenario is generated with the following initial
values:
================================
TTL_zero:False
TTL_below_threshold:False
RPF_correct_interface:False
================================
Start Executing: IP Multicast:Packet Received
 Start Point:Packet Received in Component:Router
 Stub_Construct: SRPF_Check
 Plugin: RPF_check_plugin
 Start Point:Start_RPF in Component:Router
Responsibility: packet_source_lookup in component:
Router
Responsibility: check_interf in component: Router
 OR-Fork: RFP_OF
Responsibility: drop_packet in component: Router
 End point: RPF_failure in Component:Router
 End Point: RPF_Packet_Drop part of root map
reached in Component:Router
```

(b) Multicast Forwarding Trace 2



```
This scenario is generated with the following initial values:
================================
TTL_zero:True
TTL_below_threshold:False
RPF_correct_interface:True
================================
Start Executing: IP Multicast:Packet Received
 Start Point:Packet Received in Component:Router
 Stub_Construct: SRPF_Check
 Plugin: RPF_check_plugin
 Start Point:Start_RPF in Component:Router
Responsibility: packet_source_lookup in component: Router
Responsibility: check_interf in component: Router
 OR-Fork: RFP_OF
 End point: RPF_success in Component:Router
 Stub_Construct: STTL_Check
 Plugin: TTL_plugin
 Start Point: Start_TTL in Component:Router
Responsibility:  Decrement_TTL in component: Router
 OR-Fork:  TTL_zero_OF
Responsibility: drop_TTL_zero in component: Router
 End point: TTL_drop in Component:Router
 End Point: TTL_Packet_Drop part of root map reached in
Component:Router
```

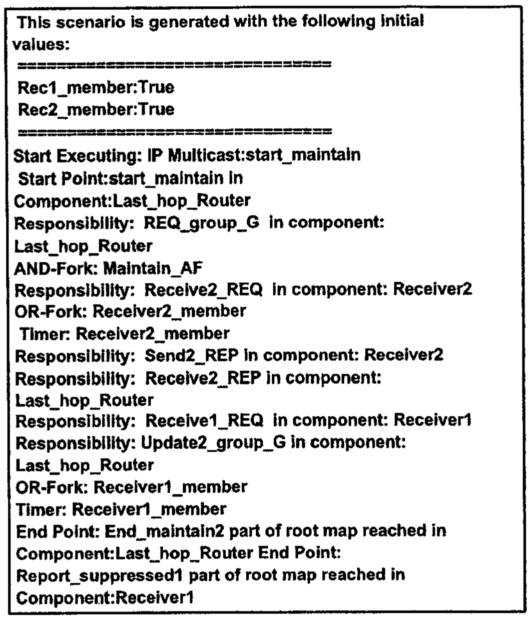
(c) Multicast Forwarding Trace 3



```
This scenario is generated with the following initial
values:
================================
TTL_zero:False
TTL_below_threshold:True
RPF_correct_interface:True
================================
Start Executing: IP Multicast:Packet Received
 Start Point:Packet Received in Component:Router
 Stub_Construct: SRPF_Check
 Plugin: RPF_check_plugin
 Start Point:Start_RPF in Component:Router
Responsibility: packet_source_lookup in component:
Router
Responsibility: check_interf in component: Router
 OR-Fork: RFP_OF
 End point: RPF_success in Component:Router
 Stub_Construct: STTL_Check
 Plugin: TTL_plugin
 Start Point: Start_TTL in Component:Router
Responsibility:  Decrement_TTL in component: Router
 OR-Fork:  TTL_zero_OF
 OR-Fork: TTL_Threshold_OF
Responsibility:  drop_below_threshold  in component:
Router
 End point: TTL_threshold_drop  in Component:Router
 End Point: TTL_th_drop part of root map reached in
Component:Router
```

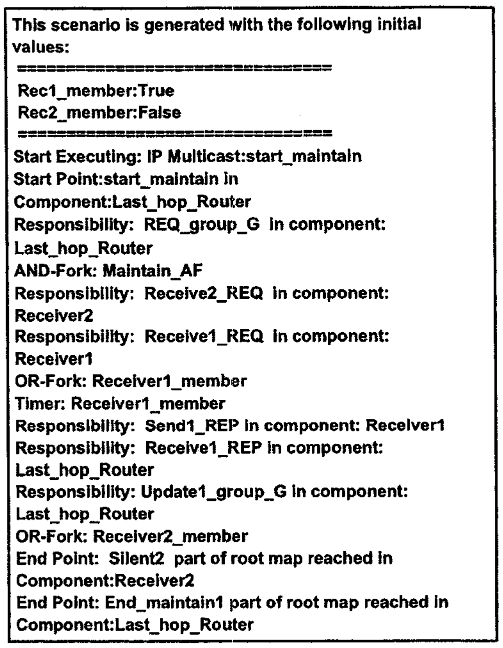
(d) Multicast Forwarding Trace 4

Figure 10.13: Multicast Forwarding Scenario Generated Traces



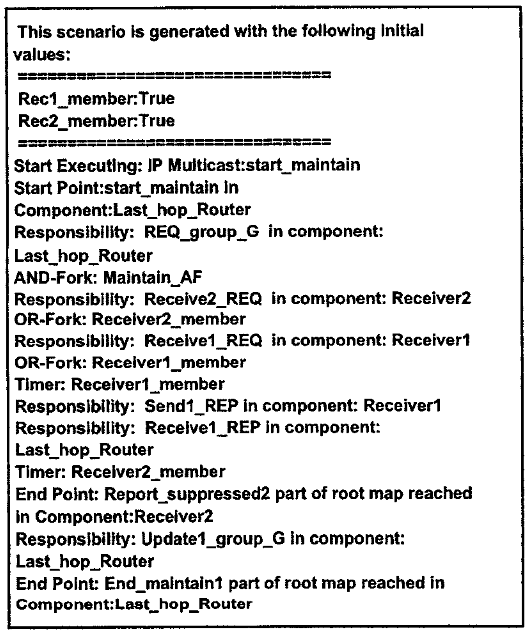
```
This scenario is generated with the following initial
values:
==============================
Rec1_member:True
Rec2_member:True
==============================
Start Executing: IP Multicast:start_maintain
Start Point:start_maintain in
Component:Last_hop_Router
Responsibility: REQ_group_G in component:
Last_hop_Router
AND-Fork: Maintain_AF
Responsibility: Receive2_REQ in component: Receiver2
OR-Fork: Receiver2_member
Timer: Receiver2_member
Responsibility: Send2_REP in component: Receiver2
Responsibility: Receive2_REP in component:
Last_hop_Router
Responsibility: Receive1_REQ in component: Receiver1
Responsibility: Update2_group_G in component:
Last_hop_Router
OR-Fork: Receiver1_member
Timer: Receiver1_member
End Point: End_maintain2 part of root map reached in
Component:Last_hop_Router End Point:
Report_suppressed1 part of root map reached in
Component:Receiver1
```

(a) Multicast Group Membership Maintenance Trace 1



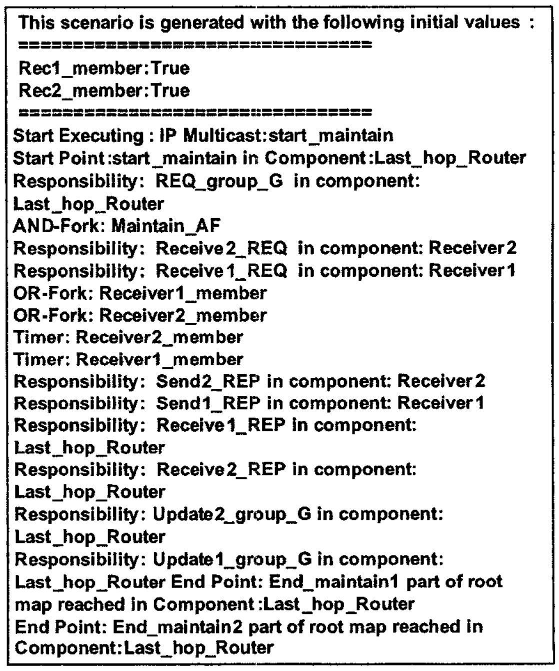
```
This scenario is generated with the following initial
values:
==============================
Rec1_member:True
Rec2_member:False
==============================
Start Executing: IP Multicast:start_maintain
Start Point:start_maintain in
Component:Last_hop_Router
Responsibility: REQ_group_G in component:
Last_hop_Router
AND-Fork: Maintain_AF
Responsibility: Receive2_REQ in component:
Receiver2
Responsibility: Receive1_REQ in component:
Receiver1
OR-Fork: Receiver1_member
Timer: Receiver1_member
Responsibility: Send1_REP in component: Receiver1
Responsibility: Receive1_REP in component:
Last_hop_Router
Responsibility: Update1_group_G in component:
Last_hop_Router
OR-Fork: Receiver2_member
End Point: Silent2 part of root map reached in
Component:Receiver2
End Point: End_maintain1 part of root map reached in
Component:Last_hop_Router
```

(b) Multicast Group Membership Maintenance Trace 2



```
This scenario is generated with the following initial
values:
==============================
Rec1_member:True
Rec2_member:True
==============================
Start Executing: IP Multicast:start_maintain
Start Point:start_maintain in
Component:Last_hop_Router
Responsibility: REQ_group_G in component:
Last_hop_Router
AND-Fork: Maintain_AF
Responsibility: Receive2_REQ in component: Receiver2
OR-Fork: Receiver2_member
Responsibility: Receive1_REQ in component: Receiver1
OR-Fork: Receiver1_member
Timer: Receiver1_member
Responsibility: Send1_REP in component: Receiver1
Responsibility: Receive1_REP in component:
Last_hop_Router
Timer: Receiver2_member
End Point: Report_suppressed2 part of root map reached
in Component:Receiver2
Responsibility: Update1_group_G in component:
Last_hop_Router
End Point: End_maintain1 part of root map reached in
Component:Last_hop_Router
```

(c) Multicast Group Membership Maintenance Trace 3



```
This scenario is generated with the following initial values :
================================
Rec1_member:True
Rec2_member:True
================================
Start Executing : IP Multicast:start_maintain
Start Point :start_maintain in Component :Last_hop_Router
Responsibility: REQ_group_G in component:
Last_hop_Router
AND-Fork: Maintain_AF
Responsibility: Receive2_REQ in component: Receiver2
Responsibility: Receive1_REQ in component: Receiver1
OR-Fork: Receiver1_member
OR-Fork: Receiver2_member
Timer: Receiver2_member
Timer: Receiver1_member
Responsibility: Send2_REP in component: Receiver2
Responsibility: Send1_REP in component: Receiver1
Responsibility: Receive1_REP in component:
Last_hop_Router
Responsibility: Receive2_REP in component:
Last_hop_Router
Responsibility: Update2_group_G in component:
Last_hop_Router
Responsibility: Update1_group_G in component:
Last_hop_Router End Point: End_maintain1 part of root
map reached in Component :Last_hop_Router
End Point: End_maintain2 part of root map reached in
Component:Last_hop_Router
```

(d) Multicast Group Membership Maintenance Trace 4

Figure 10.14: Multicast Forwarding Generated Traces

```
Start Executing : IP Multicast:start_PIM
Start Point :start_PIM in Component:DR
Responsibility:  send_Register  in component: DR
Responsibility:  decap_Register in component : RP
Responsibility: add_S_G in component: RP
AND-Fork:  PIM_AF1
End Point:  forwarded 1  part of root map reached in Component :RP
Responsibility:  send_join_S_G  in component: RP
Responsibility:  send_native_data  in component: DR
AND-Fork:  PIM_AF2
End Point:  forwarded 2 part of root map reached in Component :RP
Responsibility:  register_stop  in component: RP
End Point: stop  part of root map reached in Component :DR
```

Figure 10.15: PIM-SM Sender Registration Trace

10.1.5 UPPAAL Specification and Property Verification

IP Multicast Forwarding Scenario

Figures 10.16 and 10.17 illustrate UPPAAL specification of the UCM described in Figures 10.2, 10.4 and 10.3.

- Liveness Property: When a packet arrives to the correct interface with a TTL above the threshold and different from zero, then it should be forwarded. This property is translated into the following UPPAAL formula:

  $$(CorrectInterf \textbf{ and } !zero \textbf{ and } !below\_thr) \dashrightarrow Packet\_Forwarded\_root.end \tag{37}$$

  This property is checked to be true by the UPPAAL verifier.

- Safety Property: Packets should not be forwarded if they fail RPF check (come from a wrong interface).

  $$\textbf{A}[](!CorrectInterf) \textbf{ imply } Packet\_Forwarded\_root.end \tag{38}$$

  This property is checked to be false by the UPPAAL verifier, which is expected.

- Precedence Property: A TTL check must be preceded by an RPF check. This property is translated into the following UPPAAL formula:

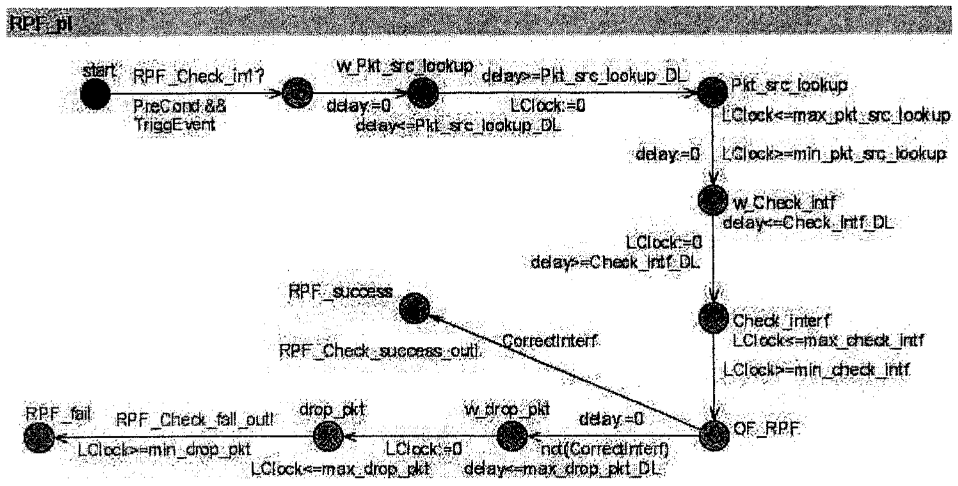
  $$\textbf{A<>}TTL\_pl.start \textbf{ imply } RPF\_pl.Check\_interf \tag{39}$$

  This property is checked to be true by the UPPAAL verifier.

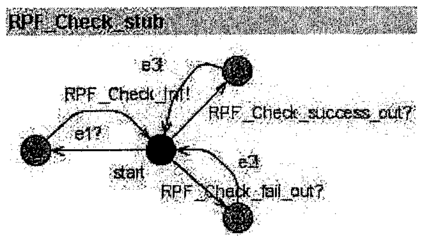
- Time Bounded Property: Packets must be forwarded within 14 time units after arrival to an ingress interface.

  $$(Packet\_rcv\_root.start) \dashrightarrow (Packet\_Forwarded\_root.end \textbf{ and } MClock <= 14) \tag{40}$$

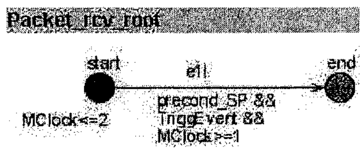
  This property is checked to be true by the UPPAAL verifier.



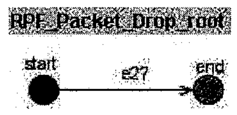
(a) RPF Plug-in Map



(b) RPF Check Stub

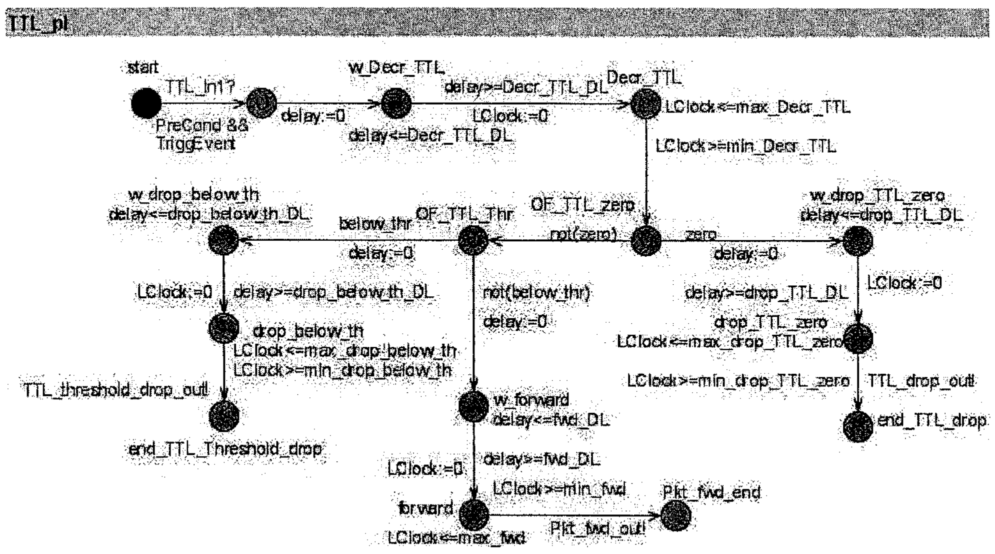


(c) Start Point Packet Received

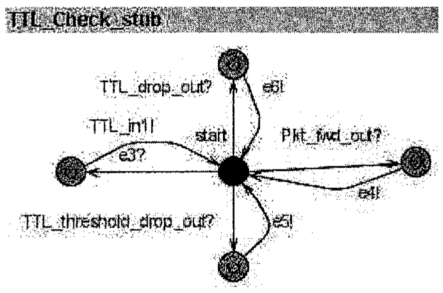


(d) End Point RPF packet drop

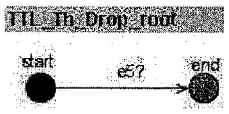
Figure 10.16: UPPAAL Implementation of IP Multicast Forwarding Scenario



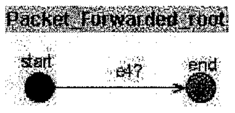
(a) TTL Plug-in Map



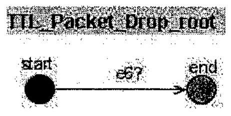
(b) TTL Check Stub



(c) End Point TTL Threshold drop



(d) End Point Packet Forwarded

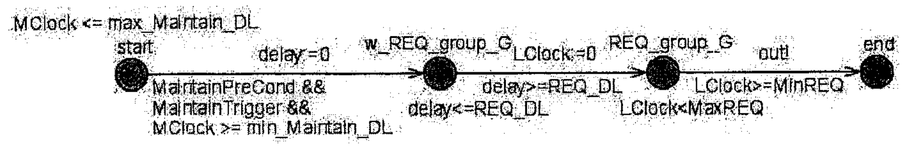


(e) End Point TTL packet drop

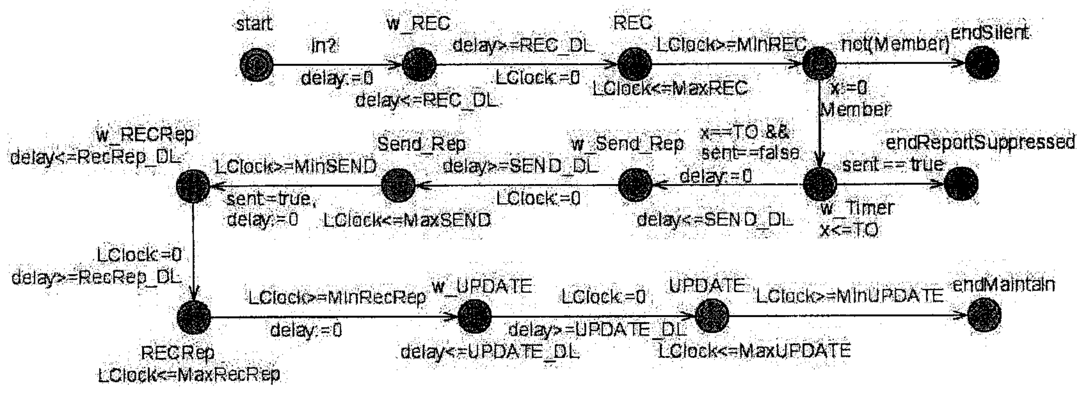
Figure 10.17: UPPAAL Implementation of IP Multicast Forwarding Scenario(2)

IP Multicast Group Membership Maintenance

Figure 10.18 shows UPPAAL specification of the UCM described in Fig 10.5. Figure 10.18(a) shows the timed automata of the segment composed of the start point *start-maintain* followed by responsibility *REQ-group-G*. Figure 10.18(b) shows the timed automata of the segment starting at responsibility *receive-REQ* and ends with the end points *Silent*, *Report-suppressed* and *End-maintain*. The later is instantiated twice, one for each receiver. We assume that both receivers are members of the same multicast group and all responsibilities have a duration between 1 and 2 with a delay of 1. The three processes are connected through an AND-Fork TA template 7.19(c).



(a) Segment1



(b) Segment2

Figure 10.18: Timed Automata of IP Multicast Group Membership Maintenance

- **Precedence Property:** For any receiver, the sending of a report is always preceded by a reception of a query. This property is translated into the following UPPAAL formula:

$$\mathbf{A}{<>}(rec1.Send\_Rep \ \mathbf{imply} \ seg1.REQ\_group\_G) \qquad (41)$$

  This property is checked to be true by the UPPAAL verifier.

- **Liveness Property:** In the presence of receivers, the multicast group should be updated. This property is translated into the following formula:

$$\mathbf{E}{<>}(rec1.UPDATE \ \mathbf{or} \ rec2.UPDATE) \qquad (42)$$

  This property holds since one of the two receivers responds to the router query and the group is eventually updated.

- **Time Bounded Property:** Sending a report occurs at least 10 time units after the start of the scenario. This property is translated into the following UPPAAL formula:

$$\mathbf{A}[\,](rec1.Send\_Rep \ \mathbf{imply} \ MClock > 10) \tag{43}$$

  This property is not satisfied and UPPAAL generates an execution trace of a counter example showing that the responsibility *send_REP* may occur as soon as *MClock* is greater or equal to 7.

- **Safety Property:** In the presence of more than one receiver, only one and only one receiver should send a report. This property is translated into the following UPPAAL formula:

$$\mathbf{A}[\,]\mathbf{not}(rec1.Send\_Rep \ \mathbf{and} \ rec2.Send\_Rep). \tag{44}$$

  This property fails leading to the generation of a counter example. This failure is due to the fact that the timer timeout event and responsibility *Send_Rep* occur in two distinct steps. Hence, timer timeouts in Receiver 1 and Receiver 2 may be triggered one after the other. In such a case, both receivers will send a report. This behavior is corrected by replacing the plain timer by a timer with action (Fig. 7.19(g)) which makes the action of sending a report part of the timeout transition. Therefore, the property becomes true.

PIM-SM Sender Registration

Figure 10.19 illustrates UPPAAL specification of the UCM described in Figure 10.9.

- **Liveness Property:** Both register packets and native data packets are forwarded. This property is translated into the following UPPAAL formula:

$$\mathbf{A}[\,](PIM\_Forwarded1\_root.end \ \mathbf{and} \ PIM\_Forwarded2\_root.end) \tag{45}$$

  This property is checked to be true by the UPPAAL verifier.

- **Response Property:** A register packet must be followed by a stop_register packet. This property is translated into the following UPPAAL formula:

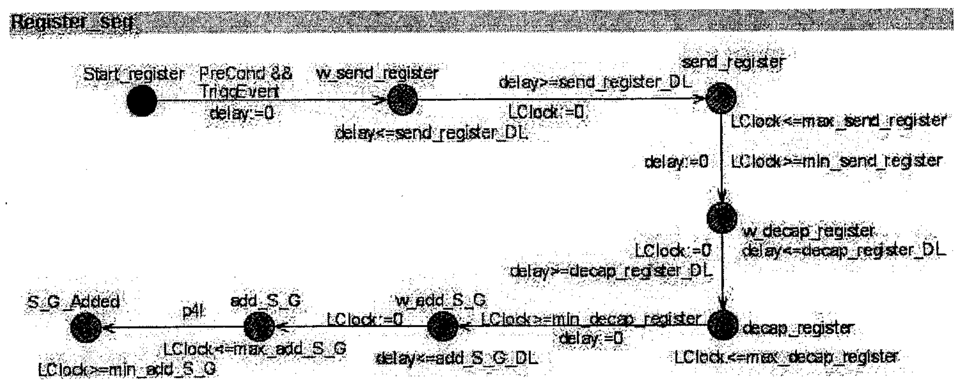
$$Register\_seg.send\_register \dashrightarrow Stop\_seg.register\_stop \tag{46}$$

  This property is checked to be true by the UPPAAL verifier.

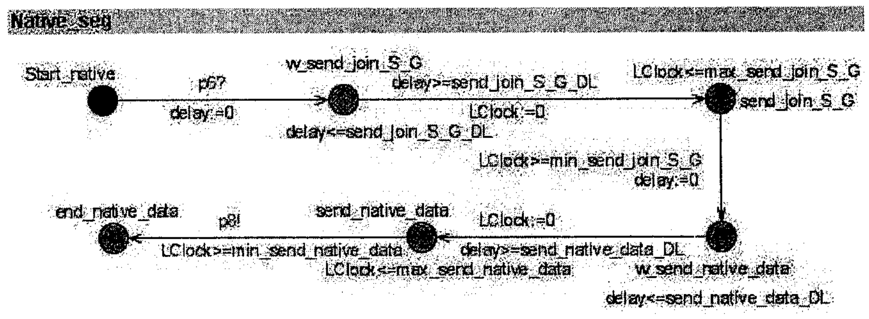
- **Time Bounded Property:** PIM-SM Sender registration scenario should not take more than 17 time units. This property is translated into the following UPPAAL formula:

$$Register\_seg.Start\_register \dashrightarrow (Stop\_seg.end\_stop \ \mathbf{and} \ MClock < 16) \tag{47}$$

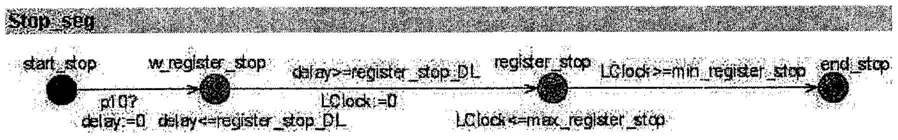
  This property is not satisfied and UPPAAL generates an execution trace of a counter example showing that this scenario takes at least 18 time units to complete.



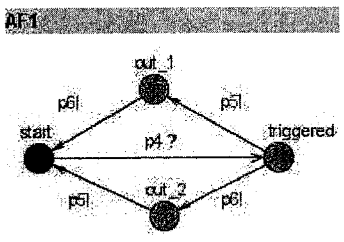
(a) Register Segment



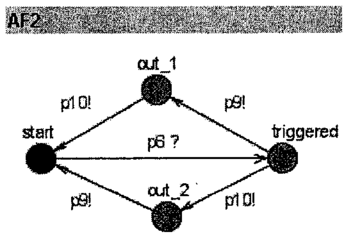
(b) Native Segment



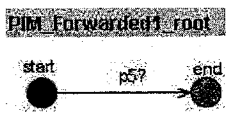
(c) Stop Segment



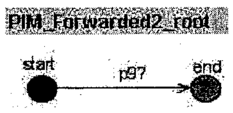
(d) AND-Fork1



(e) AND-Fork2



(f) End point Forwarded 1



(g) End point Forwarded 2

Figure 10.19: UPPAAL Implementation of PIM-SM Sender Registration Scenario

10.2 Case Study 2: Online Store

The second case study is a web application for an online store [ARW05] (named widgets.com) where users can purchase license keys for software components (or widgets).

10.2.1 Online Store: System Overview and UCM Specification

The online store implements four use cases: Browse Catalog, Checkout, Process Payment, and Download. Browse Catalog comprises selecting categories, selecting products to request product detail, adding products to a shopping cart, and editing the cart. Checkout includes signing in for an account, building an order summary, and confirming the order. Process Payment involves asking the bank to process the payment information associated with the account. Download comprises going to a download area, and downloading the purchased licenses.

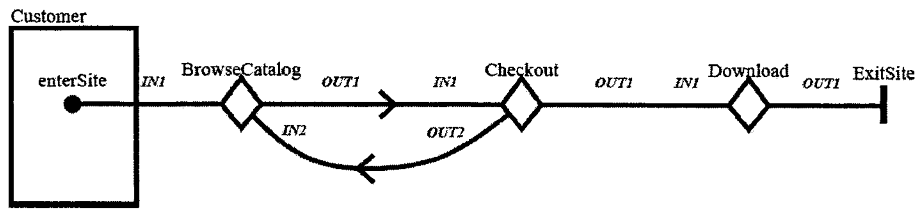


Figure 10.20: Root Map for the Widgets.com Online Store [ARW05]

Figure 10.20 shows the root map for the widgets.com applications. It contains three stubs, one for the Browse Catalog, one for the Checkout, and one for the Download use case. Figures 10.21 to 10.23 show the plug-ins for the BrowseCatalog, Checkout, and Download stubs. Start points in the Customer component correspond to events (e.g., hyperlinks and buttons) that customers can trigger. The end points correspond to page updates visible to the customers.

Selecting *goCheckout* will terminate Browse Catalog, and initiate the Checkout use case which is described in Figure 10.22. Checkout plug-in map requires the customer to input a valid account number. Once the order is confirmed, the payment done and the invoice displayed (i.e. stub ProcessPayment), the customer proceeds to the Download scenario, where the bought widgets and license keys are available for download (Figure 10.23).

Several global variables are used in this case study. They are listed in Table 10.2.1.

Preconditions were added to many start points to reflect the situations under which they can be triggered. For instance, the preconditions for the start points in the BrowseCatalog plug-in map are described in Table 10.2.1.

Several responsibilities in this case study also modify the content of these variables. Table 10.2.1 shows, for the same map, how these variables are updated by the responsibilities.

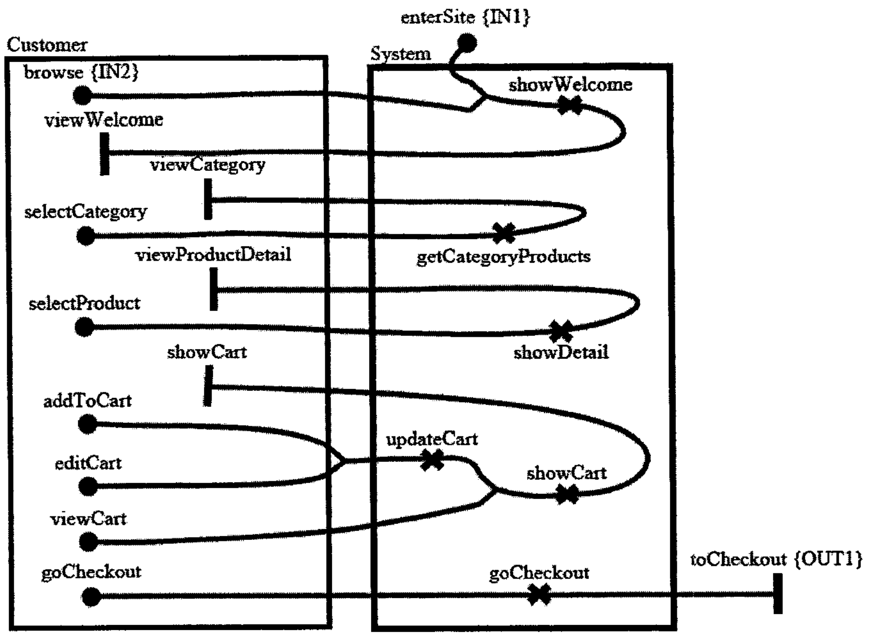


Figure 10.21: Plug-in for BrowseCatalog Stub [ARW05]

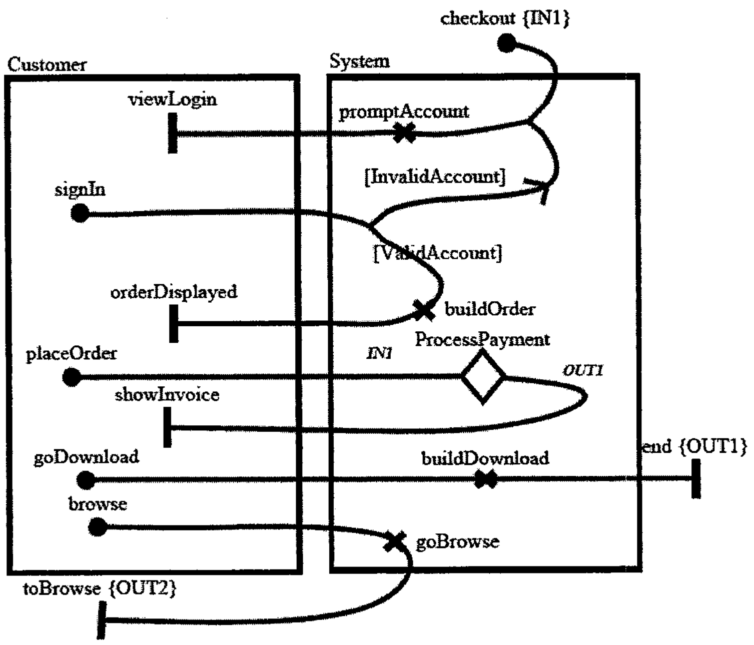


Figure 10.22: Plug-in for Checkout Stub [ARW05]

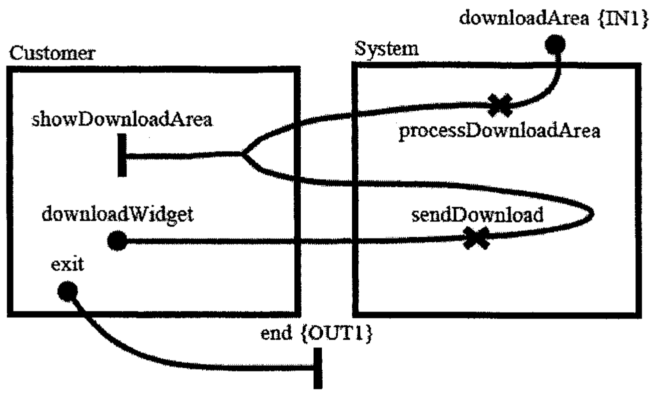


Figure 10.23: Plug-in for Download Stub [ARW05]

Variable	Description
CanAddProd	Products can be added on this page
CanGoDownload	Can go to the download area
CanPlaceOrder	An order can be placed on this page
CanSignIn	The customer can sign in
CartAvailable	The cart is visible
CategoryAvailable	Categories can be selected on this page
InBrowser	In the browser page
InCheckout	In the checkout page
InDownloadArea	In download area
ProductsDisplayed	Products are displayed
SuccessfulDownload	The download was successful
ValidAccount	The customer account is valid

Table 10.1: Online Store: Global Boolean Variables [ARW05]

Start Point	Precondition
enterSite	-
browse	InBrowser
selectCategory	InBrowser $\wedge$ CategoryAvailable
selectProduct	InBrowser $\wedge$ ProductsDisplayed
addToCart	InBrowser $\wedge$ CanAddProd
editCart	InBrowser $\wedge$ CartAvailable
viewCart	InBrowser
goCheckout	InBrowser $\wedge$ CartAvailable
downloadWidget	InDownloadArea

Table 10.2: Online Store: Start Point Preconditions for Browsecatalog Plug-in [ARW05]

Responsibility	Modification (T for True, F for False)
getCategoryProducts	ProductsDisplayed ← T
goCheckout	InBrowser ← F, CartAvailable ← F, CategoryAvailable ← F
showCart	CartAvailable ← T, CanAddProd ← F, ProductsDisplayed ← F
showDetail	CanAddProd ← T, ProductsDisplayed ← F
showWelcome	InBrowser ← T, CartAvailable ← F, CategoryAvailable ← T
promptAccount	CanSignIn ← T
buildOrder	CanSignIn ← F, CanPlaceOrder ← T
Pay	CanPlaceOrder ← F
processDownloadArea	InDownloadArea ← T
sendDownload	SuccessfulDownload ← T

Table 10.3: Online Store: Variables Modified by Responsibilities in BrowseCatalog Plug-in [ARW05]

10.2.2 AsmL Specification of the Online Store

Figure 10.24 describes the AsmL implementation of the global variable defined in the previous section.

```
var InBrowser = new BooleanValue(false) // In the browser page
var CartAvailable = new BooleanValue(false) // The cart is visible
var CategoryAvailable = new BooleanValue(false) // Categories can be selected on this page
var ProductsDisplayed = new BooleanValue(false) // Products are displayed
var CanAddProd = new BooleanValue(false) // Products can be added on this page
var CanGoDownload = new BooleanValue(false) // Can go to the download area
var CanPlaceOrder = new BooleanValue(false) // An order can be placed on this page
var CanSignIn = new BooleanValue(false) // The customer can sign in
var InCheckout = new BooleanValue(false) // In the checkout page
var InDownloadArea = new BooleanValue(false) // In download area
var SuccessfulDownload = new BooleanValue(false) // The download was successful
var ValidAccount = new BooleanValue(false) // The customer account is valid
var WidgetDownloaded = new BooleanValue(false) // Widget is downloaded
var CartUpdated = new BooleanValue(false) // Widget is downloaded
var Browse_from_checkout = new BooleanValue(false) // enables browsing from the checkout plug-in
var Max_SelectCatalog = 5
var Select_Catalog = new BooleanValue(true)
```

Figure 10.24: AsmL Implementation: Online Store Global Variables

Figures 10.25, 10.26, 10.27, 10.28 and 10.29 describes respectively the AsmL implementation of BrowseCatalog plug-in, Checkout plug-in, ProcessPayment plug-in, Download plug-in and the root map.

```
var sp_enterSite_browse as SP_Construct = SP_Construct (Browse_in1 , e2 , "sp_enterSite_browse", Boolean-
Var(pre_cond_start), Customer)
var sp_Browse as SP_Construct = SP_Construct (Browse_in2 , e3 , "sp_Browse", BooleanVar(_true) , Customer)
var showWelcome as R_Construct = R_Construct(e4 , e5, "showWelcome",System)
var viewWelcome as EP_Construct = EP_Construct(e5 , h0 , "viewWelcome", true, Customer)
var OJ_Browse as OJ_Construct = OJ_Construct(e2,e3,e4, "OJ-Browse", System)
var sp_selectCategory as SP_Construct = SP_Construct (selectCategory_in1 , e6 , "sp_selectCategory ", Boolean-
Var(InBrowser)+BooleanVar(CategoryAvailable)+ BooleanVar(Select_Catalog), Customer)
var getCategoryProducts as R_Construct = R_Construct(e6 , e7, "getCategoryProducts",System)
var viewCategory as EP_Construct = EP_Construct(e7 , h0 , "viewCategory", true, Customer)
var sp_selectProduct as SP_Construct = SP_Construct (selectProduct_in1 , e8 , "sp_selectProduct", (Boolean-
Var(InBrowser)+BooleanVar(ProductsDisplayed)), Customer)
var showDetail as R_Construct = R_Construct(e8 , e9, "showDetail",System)
var viewProductDetail as EP_Construct = EP_Construct(e9 , h0 , "viewProductDetail", true, Customer)
var sp_addToCart as SP_Construct = SP_Construct (addCart_in1 , e10 , "sp_addToCart", (Boolean-
Var(InBrowser)+BooleanVar(CanAddProd)), Customer)
var sp_editCart as SP_Construct = SP_Construct (editCart_in1 , e11 , "sp_editCart", (Boolean-
Var(InBrowser)+BooleanVar(CartAvailable)), Customer)
var sp_viewCart as SP_Construct = SP_Construct (viewCart_in1 , e12 , "sp_viewCart", Boolean-
Var(InBrowser)+BooleanVar(CartUpdated), Customer)
var OJ_add_edit as OJ_Construct = OJ_Construct(e10,e11,e13, "OJ_add_edit", System)
var OJ_add_edit_view as OJ_Construct = OJ_Construct(e12,e14,e15, "OJ_add_edit_view", System)
var updateCart as R_Construct = R_Construct(e13 , e14, "updateCart",System)
var showCart as R_Construct = R_Construct(e15 , e16, "showCart",System)
var showCart_EP as EP_Construct = EP_Construct(e16 , h0 , "showCart_EP", true, Customer)
var sp_goCheckout as SP_Construct = SP_Construct (goCheckout_in1 , e17 , "sp_goCheckout", (Boolean-
Var(InBrowser)+BooleanVar(CartAvailable)), Customer)
var goCheckout as R_Construct = R_Construct(e17 , e18, "goCheckout",System)
var toCheckout as EP_Construct = EP_Construct(e18 , h0 , "toCheckout", true, Unbound)

var BrowseCatalog_Plugin as Maps = Maps ("BrowseCatalog_plugin", {UCMElement(sp_enterSite_browse, e2
, OJ_Browse), UCMElement(sp_Browse, e3 , OJ_Browse), UCMElement(OJ_Browse, e4 , showWelcome),
UCMElement(showWelcome, e5 , viewWelcome), UCMElement(viewWelcome, h0 , viewWelcome), UCMEle-
ment(sp_selectCategory, e6 , getCategoryProducts), UCMElement(getCategoryProducts, e7 , viewCategory),
UCMElement(viewCategory, h0 , viewCategory), UCMElement(sp_selectProduct, e8, showDetail), UCMEle-
ment(showDetail, e9, viewProductDetail), UCMElement(viewProductDetail, h0 , viewProductDetail), UCMEle-
ment(sp_addToCart, e10 , OJ_add_edit), UCMElement(OJ_add_edit, e13 , updateCart), UCMElement(sp_editCart,
e11 , OJ_add_edit), UCMElement(updateCart, e14 , OJ_add_edit_view), UCMElement(sp_viewCart, e12 ,
OJ_add_edit_view), UCMElement(OJ_add_edit_view, e15 , showCart), UCMElement(showCart, e16 , show-
Cart_EP), UCMElement(showCart_EP, h0 , showCart_EP), UCMElement(sp_goCheckout, e17 , goCheckout),
UCMElement(goCheckout, e18 , toCheckout), UCMElement(toCheckout, h0 , toCheckout)})
```

Figure 10.25: AsmL Implementation: Browse Catalog Plug-in

```
var sp_checkout as SP_Construct = SP_Construct (sp_checkout_in1 , e19 , "sp_checkout", BooleanVar(InCheckout),
Customer)
var sp_signIn as SP_Construct = SP_Construct (sp_signIn_in1, e22 , "sp_signIn", Boolean-
Var(CanSignIn)+BooleanVar(InCheckout), Customer)
var sp_placeOrder as SP_Construct = SP_Construct (sp_placeOrder_in1, e27 , "sp_placeOrder", Boolean-
Var(CanPlaceOrder)+BooleanVar(InCheckout), Customer)
var sp_goDownload as SP_Construct = SP_Construct (sp_goDownload_in1, e29 , "sp_goDownload", Boolean-
Var(CanGoDownload)+BooleanVar(InCheckout), Customer)
var sp_Browse_checkout as SP_Construct = SP_Construct (sp_Browse_checkout_in1, e31 , "sp_Browse_checkout",
BooleanVar(Browse_from_checkout), Customer)
var promptAccount as R_Construct = R_Construct(e20 , e21, "promptAccount",System)
var buildOrder as R_Construct = R_Construct(e25 , e26, "buildOrder",System)
var buildDownload as R_Construct = R_Construct(e29 , e30, "buildDownload",System)
var goBrowse as R_Construct = R_Construct(e31 , e32, "goBrowse ",System)
var viewLogin as EP_Construct = EP_Construct(e21 , h0 , "viewLogin", true, Customer)
var orderDisplayed as EP_Construct = EP_Construct(e26 , h0 , "orderDisplayed", true, Customer)
var showInvoice as EP_Construct = EP_Construct(e28 , h0 , "showInvoice", true, Customer)
var end_checkout as EP_Construct = EP_Construct(e30 , h0 , "end_checkout", true, Unbound)
var toBroswe_checkout as EP_Construct = EP_Construct(e32 , h0 , "toBroswe_checkout", true, Customer)
var OJ_checkout as OJ_Construct = OJ_Construct(e19,e23,e20, "OJ_checkout", System)
var OF_checkout as OF_Construct = OF_Construct(e22,OR_Selection(e23, -BooleanVar(ValidAccount)),
OR_Selection(e25, BooleanVar(ValidAccount)), "OF_checkout", System)
var ProcessPayment as Stub_Construct= Stub_Construct(e27,e28,{Stub_Selection (ProcessPayment_Plugin,
BooleanVar(_true))}, {Stub_Binding(ProcessPayment_Plugin, e27, sp_ProcessPayment), Stub_Binding (Process-
Payment_Plugin, e28, ep_ProcessPayment)}, "ProcessPayment")
```

```
var Checkout_Plugin as Maps = Maps ("Checkout_Plugin", {UCMElement(sp_checkout, e19 , OJ_checkout),
UCMElement(OJ_checkout, e20 , promptAccount), UCMElement(promptAccount, e21 , viewLogin), UCMEle-
ment(viewLogin, h0 , viewLogin), UCMElement(sp_signIn, e22 , OF_checkout), UCMElement(OF_checkout,
e23 , OJ_checkout), UCMElement(OF_checkout, e25 , buildOrder), UCMElement(buildOrder, e26, or-
derDisplayed), UCMElement(orderDisplayed, h0, orderDisplayed), UCMElement(sp_placeOrder, e27 , Pro-
cessPayment), UCMElement(ProcessPayment, e28 , showInvoice), UCMElement(showInvoice, h0 , showIn-
voice), UCMElement(sp_goDownload, e29 , buildDownload), UCMElement(buildDownload, e30 , end_checkout),
UCMElement(end_checkout, h0 , end_checkout), UCMElement(sp_Browse_checkout, e31 , goBrowse), UCMEle-
ment(goBrowse, e32 , toBroswe_checkout), UCMElement(toBroswe_checkout, h0 , toBroswe_checkout)})
```

Figure 10.26: AsmL Implementation: Checkout Plug-in Map

```
var sp_ProcessPayment as SP_Construct = SP_Construct (sp_ProcessPayment_in1 , t6 , "sp_ProcessPayment",
BooleanVar(InCheckout), System)
var Pay as R_Construct = R_Construct(t6 , t7, "Pay",System)
var ep_ProcessPayment as EP_Construct = EP_Construct(t7 , h0 , "ep_ProcessPayment", true, System)
```

```
var ProcessPayment_Plugin as Maps = Maps ("ProcessPayment_Plugin", {UCMElement(sp_ProcessPayment, t6 ,
Pay), UCMElement(Pay, t7 , ep_ProcessPayment), UCMElement(ep_ProcessPayment, h0 , ep_ProcessPayment)})
```

Figure 10.27: AsmL Implementation: Payment Plug-in

```
var sp_downloadArea as SP_Construct = SP_Construct (sp_downloadArea_in1 , e33 , "sp_downloadArea", Boolean-
Var(CanGoDownload), System)
var sp_downloadWidget as SP_Construct = SP_Construct (sp_downloadWidget_in1 , e37 , "sp_downloadWidget",
BooleanVar(InDownloadArea)+BooleanVar(SuccessfulDownload), System)
var sp_exit as SP_Construct = SP_Construct (sp_exit_in1 , e38 , "sp_exit", BooleanVar(WidgetDownloaded), Cus-
tomer)
var processDownloadArea as R_Construct = R_Construct(e33 , e34, "processDownloadArea",System)
var sendDownload as R_Construct = R_Construct(e37 , e36, "sendDownload",System)
var OJ_download as OJ_Construct = OJ_Construct(e34,e36,e35, "OJ_download", Customer)
var showDownloadArea as EP_Construct = EP_Construct(e35 , h0 , "showDownloadArea", true, Customer)
var ep_Download as EP_Construct = EP_Construct(e38 , h0 , "ep_Download", true, Unbound)
```

```
var Download_Plugin as Maps = Maps ("Download_Plugin", {UCMElement(sp_downloadArea, e33 , processDown-
loadArea), UCMElement(processDownloadArea, e34 , OJ_download), UCMElement(OJ_download, e35 , show-
DownloadArea) , UCMElement(showDownloadArea, h0 , showDownloadArea), UCMElement(sp_downloadWidget,
e37 , sendDownload) , UCMElement(sendDownload, e36 , OJ_download), UCMElement(sp_exit, e38 ,
ep_Download), UCMElement(ep_Download, h0 , ep_Download)})
```

Figure 10.28: AsmL Implementation: Download Plug-in Map

```
var sp_enterSite_root as SP_Construct = SP_Construct (in1 , t1 , "sp_enterSite_root", BooleanVar(pre_cond_start),
Customer)
var ExitSite as EP_Construct = EP_Construct(t5 , h0 , "ExitSite ", true, Unbound)
var BrowseCatalog as Stub_Construct= Stub_Construct(t1,t3,t2,Stub_Selection (BrowseCatalog_Plugin, Boolean-
Var(_true)), Stub_Binding(BrowseCatalog_Plugin, t1, sp_enterSite_browse), Stub_Binding (BrowseCatalog_Plugin,
t2, toCheckout), Stub_Binding(BrowseCatalog_Plugin, t3, sp_Browse), "BrowseCatalog")
var CheckoutStub as Stub_Construct= Stub_Construct({t2},{t3,t4},{Stub_Selection (Checkout_Plugin,
BooleanVar(_true))}, {Stub_Binding(Checkout_Plugin, t2, sp_checkout), Stub_Binding (Checkout_Plugin, t4,
end_checkout), Stub_Binding (Checkout_Plugin, t3, toBroswe_checkout)}, "CheckoutStub")
var DownloadStub as Stub_Construct= Stub_Construct({t4},{t5},{Stub_Selection (Download_Plugin, Boolean-
Var(_true))}, {Stub_Binding(Download_Plugin, t4, sp_downloadArea), Stub_Binding (Download_Plugin, t5,
ep_Download)}, "DownloadStub")

var RootMap as Maps = Maps("RootMap", {UCMElement(sp_enterSite_root ,t1 , BrowseCatalog), UCMEle-
ment(BrowseCatalog, t2 , CheckoutStub), UCMElement(CheckoutStub, t4 , DownloadStub), UCMEle-
ment(DownloadStub, t5 , ExitSite), UCMElement(CheckoutStub, t3, BrowseCatalog), UCMElement(ExitSite, h0
, ExitSite)})
```

Figure 10.29: AsmL Implementation: Online Store Root Map

10.2.3 Online Store Scenarios Generation

The following traces are generated from the AsmL specification:

- Figure 10.30 describes one possible trace of scenario where all responsibilities are executed once and the costumer has a valid account.
- Figure 10.31 describes one possible trace of scenario where the customer selects browse from the checkout plug-in map to go again into browse Catalog plug-in map.
- Figure 10.32 describes one possible trace of scenario where the customer has an invalid account.

```
Start Executing: Online Store
Start Point:sp_enterSite_root in
Component:Customer
Stub_Construct:BrowseCatalog
Execution of Plugin: BrowseCatalog_plugin
Start Point:sp_enterSite_browse in
Component:Customer
OR-Join:OJ-Browse
Responsibility:showWelcome in component:System
End point:viewWelcome in component:Customer
Start Point:sp_selectCategory  in
Component:Customer
Responsibility:getCategoryProducts in
component:System
End point:viewCategory in component:Customer
Start Point:sp_selectProduct in
Component:Customer
Responsibility:showDetail in component:System
End point:viewProductDetail in
component:Customer
Start Point:sp_addToCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_editCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_viewCart in Component:Customer
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_goCheckout in
Component:Customer
Responsibility:goCheckout in component:System
End point:toCheckout in component:Unbound
Stub_Construct:CheckoutStub
Execution of Plugin: Checkout_Plugin
Start Point:sp_checkout in Component:Customer
OR-Join:OJ_checkout
```

```
Responsibility:promptAccount in
component:System
End point:viewLogin in component:Customer
Start Point:sp_signIn in Component:Customer
OR-Fork:OF_checkout
Responsibility:buildOrder in
component:System
End point:orderDisplayed in
component:Customer
Start Point:sp_placeOrder in
Component:Customer
Stub_Construct:ProcessPayment
Execution of Plugin: ProcessPayment_Plugin
Start Point:sp_ProcessPayment in
Component:System
Responsibility:Pay in component:System
End point:ep_ProcessPayment in
component:System
End point:showInvoice in component:Customer
Start Point:sp_goDownload in
Component:Customer
Responsibility:buildDownload in
component:System
End point:end_checkout in component:Unbound
Stub_Construct:DownloadStub
Execution of Plugin: Download_Plugin
Start Point:sp_downloadArea in
Component:System
Responsibility:processDownloadArea in
component:System
OR-Join:OJ_ download
End point:showDownloadArea in
component:Customer
Start Point:sp_downloadWidget in
Component:System
Responsibility:sendDownload in
component:System
OR-Join:OJ_download
End point:showDownloadArea in
component:Customer
Start Point:sp_exit in Component:Customer
End point:ep_Download in component:Unbound
End point:ExitSite  in component:Unbound
Execution Terminated successfully
```

Figure 10.30: Online Store: Trace 1

```
Start Executing: Online Store
Start Point:sp_enterSite_root in Component:Customer
Stub_Construct:BrowseCatalog
Execution of Plugin: BrowseCatalog_plugin
Start Point:sp_enterSite_browse in Component:Customer
OR-Join:OJ-Browse
Responsibility:showWelcome in component:System
End point:viewWelcome in component:Customer
Start Point:sp_selectCategory  in Component:Customer
Responsibility:getCategoryProducts in component:System
End point:viewCategory in component:Customer
Start Point:sp_selectProduct in Component:Customer
Responsibility:showDetail in component:System
End point:viewProductDetail in component:Customer
Start Point:sp_addToCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_editCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_viewCart in Component:Customer
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_goCheckout in Component:Customer
Responsibility:goCheckout in component:System
End point:toCheckout in component:Unbound
Stub_Construct:CheckoutStub
Execution of Plugin: Checkout_Plugin
Start Point:sp_checkout in Component:Customer
OR-Join:OJ_checkout
Responsibility:promptAccount in component:System
End point:viewLogin in component:Customer
Start Point:sp_signIn in Component:Customer
OR-Fork:OF_checkout
Responsibility:buildOrder in component:System
End point:orderDisplayed in component:Customer
Start Point:sp_placeOrder in Component:Customer
Stub_Construct:ProcessPayment
Execution of Plugin: ProcessPayment_Plugin
Start Point:sp_ProcessPayment in Component:System
Responsibility:Pay in component:System
End point:ep_ProcessPayment in component:System
End point:showInvoice in component:Customer
Start Point:sp_Browse_checkout in Component:Customer
Responsibility:goBrowse  in component:System
End point:toBroswe_checkout in component:Customer
Stub_Construct:BrowseCatalog
Execution of Plugin: BrowseCatalog_plugin
Start Point:sp_Browse in Component:Customer
OR-Join:OJ-Browse
Responsibility:showWelcome in component:System
End point:viewWelcome in component:Customer
```

```
Start Point:sp_selectCategory  in Component:Customer
Responsibility:getCategoryProducts in component:System
End point:viewCategory in component:Customer
Start Point:sp_selectProduct in Component:Customer
Responsibility:showDetail in component:System
End point:viewProductDetail in component:Customer
Start Point:sp_addToCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_editCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_viewCart in Component:Customer
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_goCheckout in Component:Customer
Responsibility:goCheckout in component:System
End point:toCheckout in component:Unbound
Stub_Construct:CheckoutStub
Execution of Plugin: Checkout_Plugin
Start Point:sp_checkout in Component:Customer
OR-Join:OJ_checkout
Responsibility:promptAccount in component:System
End point:viewLogin in component:Customer
Start Point:sp_signIn in Component:Customer
OR-Fork:OF_checkout
Responsibility:buildOrder in component:System
End point:orderDisplayed in component:Customer
Start Point:sp_placeOrder in Component:Customer
Stub_Construct:ProcessPayment
Execution of Plugin: ProcessPayment_Plugin
Start Point:sp_ProcessPayment in Component:System
Responsibility:Pay in component:System
End point:ep_ProcessPayment in component:System
Online Store:UpStub executed
End point:showInvoice in component:Customer
Start Point:sp_goDownload in Component:Customer
Responsibility:buildDownload in component:System
End point:end_checkout in component:Unbound
Stub_Construct:DownloadStub
Execution of Plugin: Download_Plugin
Start Point:sp_downloadArea in Component:System
Responsibility:processDownloadArea in component:System
OR-Join:OJ_download
End point:showDownloadArea in component:Customer
Start Point:sp_downloadWidget in Component:System
Responsibility:sendDownload in component:System
OR-Join:OJ_download
End point:showDownloadArea in component:Customer
Start Point:sp_exit in Component:Customer
End point:ep_Download in component:Unbound
End point:ExitSite  in component:Unbound
Execution Terminated successfully
```

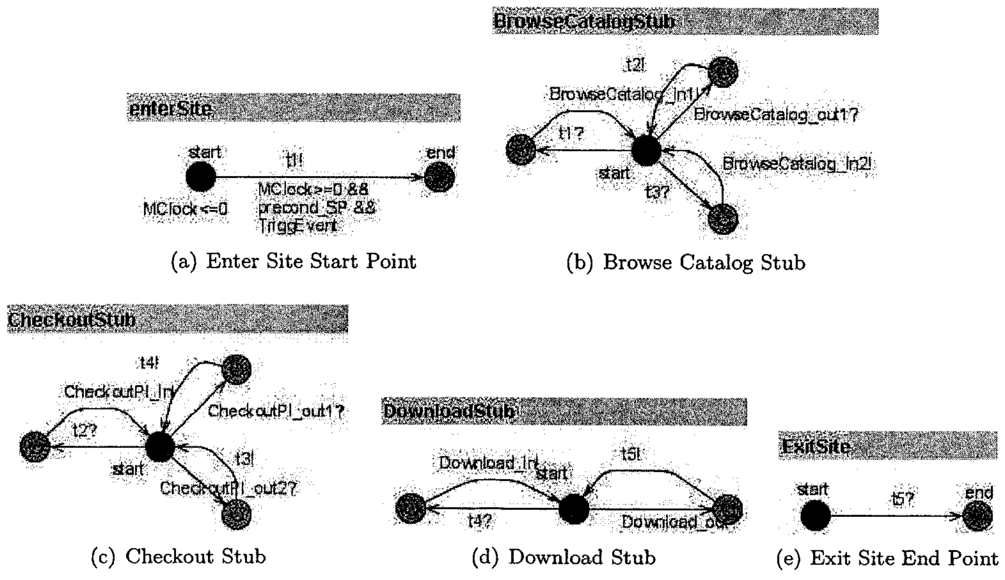
Figure 10.31: Online Store: Trace 2

```
Start Executing: Online Store
Start Point:sp_enterSite_root in Component:Customer
Stub_Construct:BrowseCatalog
Execution of Plugin: BrowseCatalog_plugin
Start Point:sp_enterSite_browse in Component:Customer
OR-Join:OJ-Browse
Responsibility:showWelcome in component:System
End point:viewWelcome in component:Customer
Start Point:sp_selectCategory  in Component:Customer
Responsibility:getCategoryProducts in component:System
End point:viewCategory in component:Customer
Start Point:sp_selectProduct in Component:Customer
Responsibility:showDetail in component:System
End point:viewProductDetail in component:Customer
Start Point:sp_addToCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_editCart in Component:Customer
OR-Join:OJ_add_edit
Responsibility:updateCart in component:System
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_viewCart in Component:Customer
OR-Join:OJ_add_edit_view
Responsibility:showCart in component:System
End point:showCart_EP in component:Customer
Start Point:sp_goCheckout in Component:Customer
Responsibility:goCheckout in component:System
End point:toCheckout in component:Unbound
Stub_Construct:CheckoutStub
Execution of Plugin: Checkout_Plugin
Start Point:sp_checkout in Component:Customer
OR-Join:OJ_checkout
Responsibility:promptAccount in component:System
End point:viewLogin in component:Customer
Start Point:sp_signIn in Component:Customer
OR-Fork:OF_checkout
OR-Join:OJ_checkout
Responsibility:promptAccount in component:System
End point:viewLogin in component:Customer
```

Figure 10.32: Online Store: Trace 3

10.2.4 UPPAAL Specification and Property Verification

Figure 10.33 illustrates UPPAAL specification of the root map for widgets.com online store.



(a) Enter Site Start Point

(b) Browse Catalog Stub

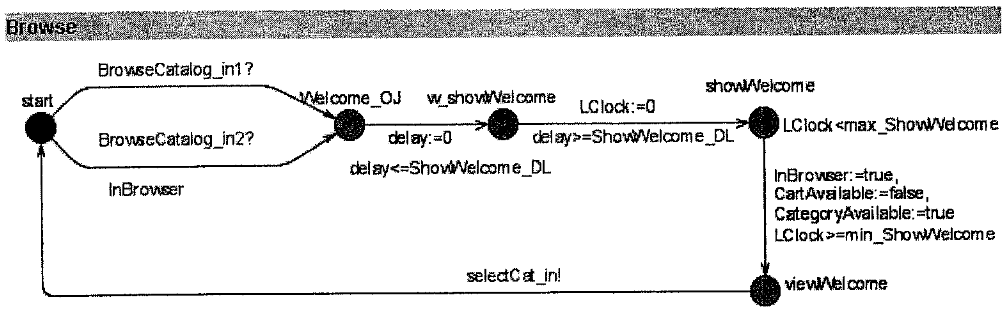
(c) Checkout Stub

(d) Download Stub

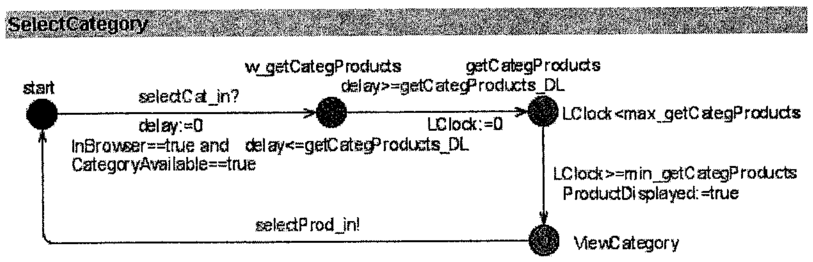
(e) Exit Site End Point

Figure 10.33: UPPAAL Implementation of Root map for Widgets.com Online Store

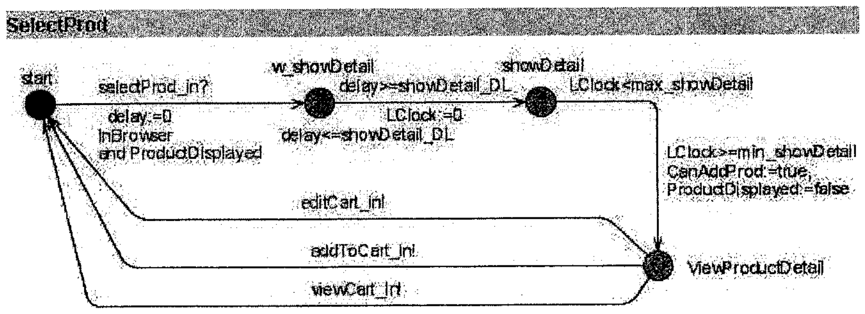
Figures 10.34 and 10.35 illustrate UPPAAL specification of the BrowseCatalog plug-in.
Figure 10.36 illustrates UPPAAL specification of the Checkout plug-in.
Figure 10.37 illustrates UPPAAL specification of the Download plug-in.



(a) Browse Segment

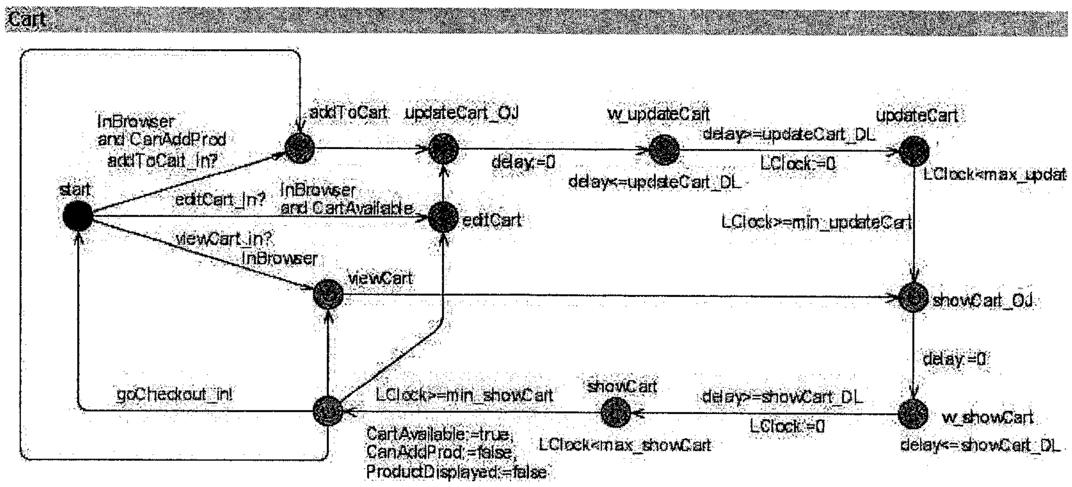


(b) Select Category Segment

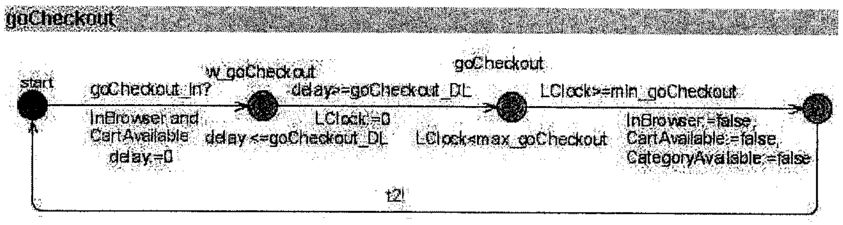


(c) Select Product Segment

Figure 10.34: UPPAAL Implementation of BrowseCatalog Plug-in

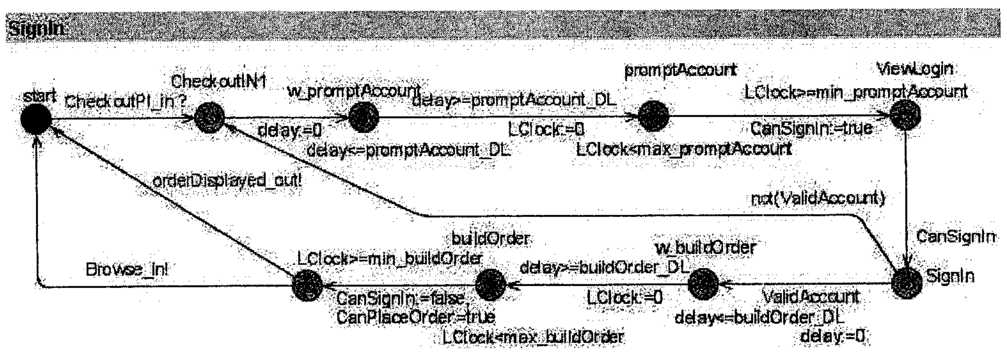


(a) Cart Segment

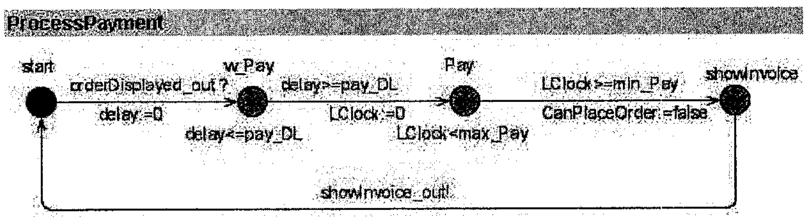


(b) goCheckout Segment

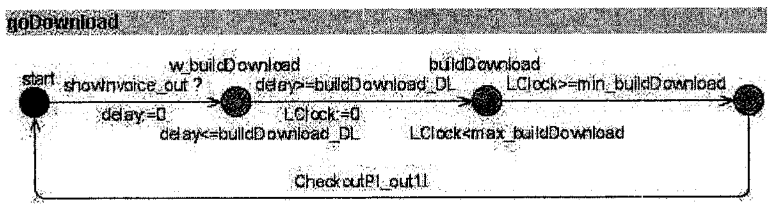
Figure 10.35: UPPAAL Implementation of BrowseCatalog Plug-in(2)



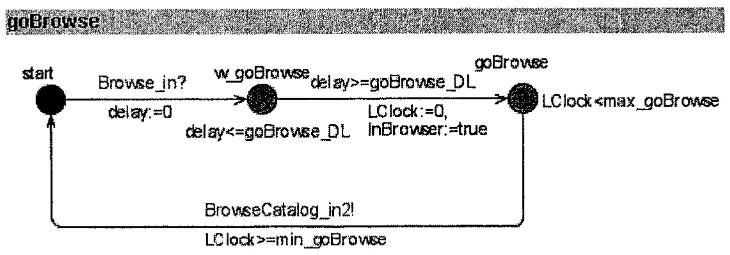
(a) SignIn Segment



(b) Place Order Segment

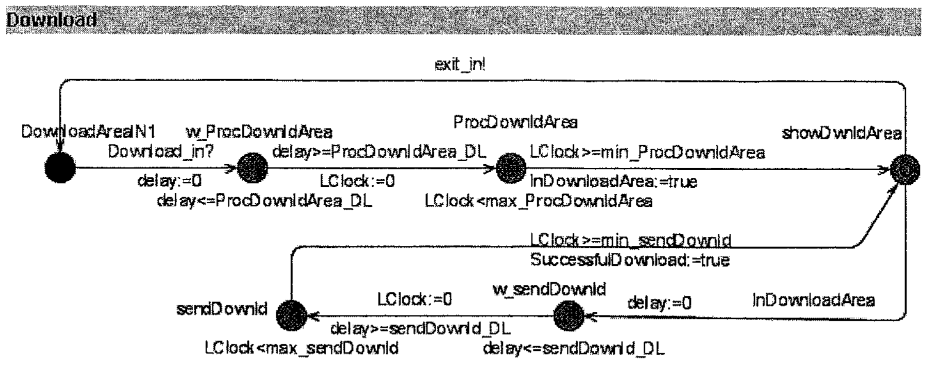


(c) goDownload Segment

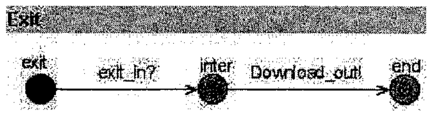


(d) goBrowse Segment

Figure 10.36: UPPAAL Implementation of Checkout Plug-in



(a) Download Segment



(b) Exit Segment

Figure 10.37: UPPAAL Implementation of Download Plug-in

For the purpose of verifying timed properties, we assume that all responsibilities have a delay equal to 1 and a duration between 1 and 2 time units.

- **Precedence Property:** Adding a product to cart must be preceded by viewing the product details. This property is translated into the following UPPAAL formula:

$$\mathbf{A}{<>}\ Cart.addToCart\ \mathbf{imply}\ SelectProd.ViewProductDetail \qquad (48)$$

  This property is checked to be true by the UPPAAL verifier.

- **Response Property:** A payment is followed by a widget download (i.e. responsibility *send-Downld*). This property is translated into the following UPPAAL formula:

$$ProcessPayment.Pay\ \dashrightarrow\ Download.sendDownld \qquad (49)$$

  This property is checked to be false by the UPPAAL verifier. Indeed, nothing can force the customer to exit the download area even after paying for the product. UPPAAL generates an execution trace of a counter example showing that after reaching location *showdwnldArea* there is a possible transition to the exit segment. This is the intended behavior not a design flaw.

- **Fairness Property:** There is a scenario where the customer selects a product, pays for it and proceeds to a download. This property is translated into the following UPPAAL formula:

$$\mathbf{A}{<>}\ Download.showDwnldArea \qquad (50)$$

This property is not satisfied because of UPPAAL's lack of fairness. After signing in, a customer can go back to browse plug-in without placing an order. UPPAAL has no means to exit a loop and to force the execution of leftover behavior.

- Safety Property: Customers with invalid accounts cannot proceed with a payment. This property is translated into the following UPPAAL formula:

$$\mathbf{A}{<>}!ValidAccount \ \mathbf{imply}\ ProcessPayment.Pay \tag{51}$$

  This property is not satisfied and this is expected.

- Time Bounded Property: The welcome screen appears within 3 time units after the customer enters the site.

$$enterSite.start \dashrightarrow Browse.viewWelcome\ \mathbf{and}\ MClock < 3 \tag{52}$$

  Having a delay equal to one time unit for responsibility *showWelcome* and a duration between 1 and 2, this property is checked to be true by the UPPAAL verifier.

10.3 Lessons Learned

In the preceding sections, we have applied our methodology to two case studies, one case study of a telecommunication protocol and one case study of web application. The model construction process in both cases was very instructive. We gained insights in various aspects of system modeling, scenario integration, dealing with hierarchical specification and property specification. From these experiments, we were able to distill some useful rules:

- **Formal modeling.** The modeling phase is very important and as useful as the validation and verification phase. As many before us, we experienced that the construction of a formal model of the system typically allows for the comprehension of the system and may expose weaknesses and inconsistencies in the design of the system.
  - **AsmL Models:** The use of AsmL as target implementation language helped to capture many specification details such as variables values, components traversed, conditions computation (i.e. logical conditions on OR-Forks and plug-in selection policies at dynamic stubs) and timing information (for timed UCM models). Depending on the user needs, more specific information can be retrieved from an UCM-AsmL specification. Such information may involve the UCM component dependencies (backward and forward component dependencies), number of times a loop is traversed, the number of times a specific plug-in is selected, etc.
  - **Timed Automata Models:** Our optimized timed automata model involves the sequential composition of UCM constructs to form a set of parallel processes. Hence, many intermediate locations can be suppressed allowing for a reduced state space and more efficient verification step. When UCM specification contains loops, TA processes may be

executed multiple times. To avoid deadlock situations, we add a transition between the end and start location in each process involved in loops

- **Property Specification.** A model checker is used to check whether a property ϕ holds for a model M. The total set of properties S which the model M should satisfy is important as it constitutes the *contract* that the model M should satisfy. Theoretically, the set S should be defined before the actual verification with a model checker is started and should not be formulated on-the-fly during the verification process. In practice, when specifying TCTL properties in UPPAAL, the process name should be appended to the location name (for instance A[] ProcessName.locationName, etc.). However, the process name is only known after the sequential composition step.

- **Reduction Techniques:** The purpose of using reduction techniques in Early Stages V&V is two-fold. First, it reduces the size of the specification (in both AsmL and TA formalisms) allowing for a more focuses analysis. Second, UCM-based change impact analysis benefit from reduction techniques to assess the impact of a change when the specification is modified to fix a design flaw or to perform an upgrade.

Chapter 11

Conclusions and Future Work

In this chapter, we first review the main contributions of the thesis, relate them to the research hypothesis and discuss whether our initial goals have been met. In Section 11.2, we briefly compare *Early Stages V&V* methodology to SPEC-VALUE and we provide insights on how both methodologies can be integrated. Finally, we propose some directions for future research.

11.1 Hypothesis and Contributions of the Thesis

This thesis presents *Early Stages V&V*, which is a methodology for the validation and verification of specifications at early stages of development process. Our approach combines formal specification techniques (FSTs) with the semi-formal language Use Case Maps. Our first research hypothesis is denoted as follows:

> *At the early stages of system development, requirements described using the Use Case Map language can be formalized in terms of Abstract State Machines (ASM). Hence, UCM models can be validated through simulation and functional testing.*

Chapter 4 presents a formal syntax and a formal operational semantics for the Use Case Maps language based on Abstract State Machines. Two possible ASM-based solutions were proposed: (1) Multi Agent ASM solution and (2) Single agent ASM solution with non deterministic interleaving. Both solutions are implemented within the ASM-UCM simulation engine (see Section 4.3), designed for simulating and executing UCM specifications.

Different theories and techniques are involved in the support of the *Early Stages V&V* methodology. Trace generation, reduction techniques (Slicing) and model checking were successfully applied in our methodology.

Our second research hypothesis is denoted as follows:

> *In the process of verifying complex systems, requirements described using the Use Case Map language can be validated and verified efficiently through the use of reduction techniques.*

Indeed, we showed that the use of reduction techniques at Use Case Maps abstraction level can help reduce the specification size, allowing for more efficient validation and verification. In particular Chapter 5 describes how slicing (i.e. both backward and forward slicing) is applied to efficiently validate a UCM specification and to assess the impact of a change (i.e. for instance a bug fix or an upgrade).

The need to incorporate non-functional aspects, and in particular time-related aspects into requirement languages has been widely recognized. We believe also that Use Case Maps notation can be extended to cover non-functional requirements such as timing constraints.

Our third research hypothesis is denoted as follows:

Use Case Maps notation can be extended to cover non-functional requirements such as timing constraints. Timed UCM can be formalized in terms of CTS (Clocked Transition System) and Timed Automata (TA) formalisms that can be analyzed and verified.

In Chapter 6, we have proposed a collection of eleven criteria that we have used to categorize and compare thirteen timed scenario notations. In Chapter7, we have extended the Use Case Maps language with time. Two formalization approaches for timed UCM language were presented: (1) Clocked Transition System (CTS) based semantics for both interleaving and true concurrency models (2) Timed Automata (TA) based semantics. Chapter 8 proposes an approach to formally verify timed UCM specifications using model checking.

These three research hypothesis were validated through the theoretical framework supporting the used techniques (i.e. ASM, CTS and TA formalisms) and through the successful application of this methodology to the simple telephony system (our running case study) and two case studies presented in Chapter 10.

Although there exists a significant body of research in the area of formal verification and model checking tools of software and hardware systems, there has only a limited industry and end-user acceptance of these tools. Beside the technical problem of state space explosion, one of the main reasons for this limited acceptance is the unfamiliarity of users with the required specification notation. We believe that Use Case Maps can be used to describe a set of commonly used properties that are presented in terms of occurrence, ordering and temporal scopes of actions. Furthermore, UCM also supports the description of properties with respect to their architectural scope. This may be achieved through a minimal extension of UCM language. Our fourth research hypothesis is denoted as follows:

The visual and easy to learn syntax of UCM, can support the description of a large set of high level properties without the need for temporal logic formalisms.

Chapter 9 has yielded two main contributions. First, we have presented a UCM based specification pattern that can simplify this activity and make it available to the novice practitioner. The specification pattern system uses templates to cover most common expected properties found in requirements specifications. We have provided a mapping of our UCM-based system to popular temporal logics CTL and TCTL. These templates combine qualitative, real-time and architectural properties into a single graphical representation. To the best of our knowledge, no existing pattern system has considered these three scopes together. Second, we have extended the traditional real-time temporal logics to include architectural aspects. We have provided an overview of the semantics of the systems targeted by what we call 'Architectural real-time temporal logic'. Moreover, we have provided formal syntax and semantics of *ArTCTL*, an extension of TCTL with architectural aspects. We believe that having the requirement specification and properties described using the same formalism will enable greater degrees of analysis while preserving a high level of abstraction.

11.2 Integrating Early Stages V&V with SPEC-VALUE

When it comes to the capture, specification, validation and verification of requirements and high-level designs, many existing methodologies can benefit from *Early Stages V&V* techniques and vice versa. In this section, we focus on a potential integration with SPEC-VALUE approach since both are based on Use Case Maps.

Early Stages V&V and SPEC-VALUE [Amy01a] share the following similarities:

- Requirements are captured with Use Case Maps, which visually describe causal scenarios bound to component structures.
- Both are iterative and incremental scenario-driven approaches.
- Mapping UCM specifications to a formal language. SPEC-VALUE uses LOTOS as formal specification technique whereas Early stages V&V uses Abstract State Machines, Clocked Transition System and Timed automata.
- Both methodologies use simulation as part of their validation step.

However, there are some differences between the two methodologies:

- Early Stages V&V is based on simulation and trace generation techniques while SPEC-VALUE uses testing as validation approach supported by a well-established LOTOS theory and a rich set of testing tools.
- Early Stages V&V assumes that the UCM system specification is available and ready for validation, whereas SPEC-VALUE provides several guidelines for the use of the UCM language and the integration of scenarios. One of SPEC-VALUE contribution is the separation of concerns between system functionalities and underlying structure.
- Early Stages V&V can be applied to both untimed and timed Use Case Maps versions, whereas SPEC-VALUE targets untimed UCM specifications only.
- Early Stages V&V integrates both validation and verification in a single methodology. Indeed, it combines simulation and model checking in order to gain confidence in the correctness of UCM models.

- SPEC-VALUE does not address the issue of requirement modification (step 8 in Figure 3.1). *Early Stages V&V* combines reduction techniques with change impact analysis to ensure that a specification change does not introduce new issues.

Looking at the underlined similarities and differences between our approach and SPEC-VALUE, we can conclude that both methodologies can potentially benefit from each other. In what follows, we provide some insights on how these two methodologies can be integrated:

- The initial step of our approach (from requirement to UCM specification) can benefit from steps 1, 2, 3 and 4 of SPEC-VALUE (see Figure 3.1) to build the UCM specifications from functional scenarios and architecture.

- SPEC-VALUE provides guidelines on the generation of generic test suites from UCM functional scenarios (step 5 in SPEC-VALUE). Eight UCM-oriented testing patterns were proposed to cover alternatives, concurrent paths, loops, multiple start points, single stubs, and causally linked stubs. The resulting test suites may be used to validate the AsmL model against the requirements.

- Both LOTOS and ASM provide a formal framework for validating specification. The user may use either language as target FDT depending on his/her familiarity the language concepts, tool support and his/her ultimate goal. If code generation is the ultimate goal, ASM represents the best choice.

- SPEC-VALUE deals only with untimed models and does not consider formal verification. Our approach introduces the notion of time in Use Case Maps and allows for the verification of timed/untimed UCM models using UPPAAL model checker.

- SPEC-VALUE may benefit from the UCM based change impact analysis (and ripple effect analysis) when it comes to the modification of the original specification. We have showed that this can be done through the use of reduction techniques at the UCM level.

11.3 Future Work

Many items left for future work are distributed among the previous chapters. The following list recalls the most important ones, which target the full automation, optimization and generalization of this work.

- Automated generation of AsmL specification from UCM model that would implement the signature of UCM constructs along with the transition relation (as discussed in Section 4.1.2).

- We will investigate the use of dynamic slicing that may significantly reduce the size of a UCM slice. Providing inputs helps reducing the domain of the UCM and only the parts that comply with the input values are kept in the final slice. Consequently, the reachability expression is also reduced.

- Automated generation of TA specification from UCM models.
- Evaluate the completeness and the effectiveness of our UCM-based pattern system by surveying real-world specifications.
- Define a complete formal semantics for architectural real-time temporal logic and investigate the integration of architectural aspects into existing model checking algorithms.

Bibliography

[AA99] D. Amyot and R. Andrade. Description of wireless intelligent network services with Use Case Maps. In *SBRC'99: 17th Brazilian Symposium on Computer Networks, Salvador, Brazil*, 1999.

[AB93] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society.

[AB02] Telelogic AB. Telelogic ab: Doors/ers, 2002.

[ABBL95] Daniel Amyot, Francis Bordeleau, Raymond J. A. Buhr, and Luigi Logrippo. Formal support for design techniques: A timethreads-LOTOS approach. In *FORTE*, pages 57–72, 1995.

[ABKO04] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 168–177, Washington, DC, USA, 2004. IEEE Computer Society.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *LICS:Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 414–425, 1990.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993.

[ACLdSS04] Ludovic Apvrille, Jean-Pierre Courtiat, Christophe Lohr, and Pierre de Saqui-Sannes. TURTLE: A real-time UML profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering*, 30(7):473–487, 2004.

[AD90] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[AE03] Daniel Amyot and Armin Eberlein. An evaluation of scenario notations and construction approaches for telecommunication systems development. *Telecommun. Syst.*, 24(1):61–94, 2003.

[AEG+98] M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, and K. Weidenhaupt. Survey on the Scenario Use in Twelve Selected Industrial Projects. Technical Report AIB-07-1998, RWTH Aachen, 1998.

[AEKN01] Nina Amla, E. Allen Emerson, Robert P. Kurshan, and Kedar S. Namjoshi. Rtdt: A front-end for efficient model checking of synchronous timing diagrams. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 387–390, London, UK, 2001. Springer-Verlag.

[AEN99] Nina Amla, E. Allen Emerson, and Kedar S. Namjoshi. Efficient decompositional model checking for regular timing diagrams. In *Conference on Correct Hardware Design and Verification Methods*, pages 67–81, 1999.

[AFH94] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. A determinizable class of timed automata. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 1–13, London, UK, 1994. Springer-Verlag.

[AGG+98] Alfred V. Aho, Sean Gallagher, Nancy D. Griffeth, Cynthia Schell, and Deborah Swayne. SCF3/Sculptor with Chisel: Requirements engineering for communications services. In *FIW*, pages 45–63, 1998.

[AH90] Rajeev Alur and Thomas A. Henzinger. Real-Time Logics: Complexity and Expressiveness. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401, Washington, D.C., 1990. IEEE Computer Society Press.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106, London, UK, 1992. Springer-Verlag.

[AH97] R. Alur and Thomas A. Henzinger. Real-time system = discrete system + clock variables. Technical Report UCB/ERL M97/78, EECS Department, University of California, Berkeley, 1997.

[AHHC03] Daniel Amyot, Xiangyang He, Yong He, and Dae Yong Cho. Generating scenarios from use case map specifications. In *QSIC '03: Proceedings of the Third International Conference on Quality Software*, page 108, Washington, DC, USA, 2003. IEEE Computer Society.

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for Message Sequence Charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.

[AIP06] M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 21–28, New York, NY, USA, 2006. ACM Press.

[ALBG99] D. Amyot, L. Logrippo, R. J. A. Buhr, and T. Gray. Use Case Maps for the capture and validation of distributed systems requirements. In *RE'99: Fourth IEEE International Symposium on Requirements Engineering*, pages 44–53, 1999.

[All81] James F. Allen. An interval-based representation of temporal knowledge. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81), Vancouver, BC, Canada, August 1981*, pages 221–226. William Kaufmann, 1981.

[ALSS$^+$06] Apvrille, Ludovic, Saqui-Sannes, Pierre, Khendek, and Ferhat. TURTLE-P: a UML profile for the formal validation of critical and distributed systems. *Software and Systems Modeling (SoSyM)*, 5(4):449–466, December 2006.

[Alu92] Rajeev Alur. *Techniques for automatic verification of real-time systems.* PhD thesis, Stanford University, Stanford, CA, USA, 1992.

[ALW05] Daniel Amyot, Luigi Logrippo, and Michael Weiss. Generation of test purposes from Use Case Maps. *Comput. Networks*, 49(5):643–660, 2005.

[AM01a] Daniel Amyot and Andrew Miga. Use Case Maps document type definition 0.23. working document. http://jucmnav.softwareengineering.ca/twiki/bin/view/UCM/UCMNavXML/, 2001.

[AM01b] Daniel Amyot and Gunter Mussbacher. Bridging the requirements/design gap in dynamic systems with Use Case Maps (UCMs). In *International Conference on Software Engineering*, pages 743–744, 2001.

[AMM02] Daniel Amyot, Nikolai Mansurov, and Gunter Mussbacher. Understanding existing software with Use Case Maps scenarios. In *Telecommunications and beyond: The Broader Applicability of SDL and MSC, Third International Workshop, SAM 2002, Aberystwyth, UK, June 24-26. Revised Papers*, pages 124–140, 2002.

[AMPF07] Charles André, Frédéric Mallet, and Marie-Agnès Peraldi-Frati. Multiform time in uml for real-time embedded applications. In *RTCSA: 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), 21-24 August 2007, Daegu, Korea*, pages 232–240, 2007.

[Amy94] Daniel Amyot. Formalization of timethreads using LOTOS. Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, 1994.

[Amy01a] Daniel Amyot. *Specification and validation of telecommunications systems with Use Case Maps and LOTOS.* PhD thesis, University of Ottawa, Ottawa, Ontario, Canada, 2001. Adviser-Luigi Logrippo.

[Amy01b] Daniel Amyot. Use Case Maps as a feature description notation. In Stephen T. Gilmore and Mark D. Ryan, editors, *Language Constructs for Describing Features*, pages 27–44, Berlin, Germany, January 2001. Springer-Verlag.

[And00] R. Andrade. Applying Use Case Maps and formal methods to the development of wireless mobile ATM networks, 2000.

[Anl00] Matthias Anlauff. XASM - an extensible, component-based ASM language. In *ASM '00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 69–90, London, UK, 2000. Springer-Verlag.

[Are02] Demissie B. Aredo. A framework for semantics of uml sequence diagrams in pvs. *J. UCS*, 8(7):674–697, 2002.

[Arn96] Robert S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[ARW05] Daniel Amyot, Jean-François Roy, and Michael Weiss. UCM-driven testing of web applications. In *SDL Forum*, pages 247–264, 2005.

[ASM06] ASML. Microsoft research: The abstract state machine language. http://research.microsoft.com/foundations/AsmL/, 2006.

[BAL97] Hanene Ben-Abdallah and Stefan Leue. Expressing and analyzing timing constraints in Message Sequence Chart Specifications. Technical Report 97-04, Department of Electrical and Computer Engineering, University of Waterloo, 1997.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[BC96] R. J. A. Buhr and R. S. Casselman. *Use case maps for object-oriented systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[BCR00a] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. An ASM semantics for UML activity diagrams. In *AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 293–308, London, UK, 2000. Springer-Verlag.

[BCR00b] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the dynamics of UML state machines. In *ASM'00: Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, pages 223–241, London, UK, 2000. Springer-Verlag.

[BDM02] Simona Bernardi, Susanna Donatelli, and Jos&#233; Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 35–45, New York, NY, USA, 2002. ACM Press.

[BdP00] Karin Breitman and Julio Cesar Sampaio do Prado. Scenario evolution: A closer view on relationships. In *ICRE '00: Proceedings of the 4th International Conference on Requirements Engineering (ICRE'00)*, pages 95–105, Washington, DC, USA, 2000. IEEE Computer Society.

[BEGM98] Raymond J. A. Buhr, M. Elammari, Tom Gray, and Serge Mankovski. Applying Use Case Maps to multi-agent systems: A feature interaction example. In *HICSS (6)*, pages 171–179, 1998.

[Ber03] Kirsten Berkenkotter. Using UML 2.0 in real-time development: A critical review. In *Workshop Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS'03) at UML'03*, pages 41–54, San Francisco, CA, USA, October 2003.

[BF99] Victor A. Braberman and Miguel Felder. Verification of real-time designs: combining scheduling theory with automatic formal verification. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 494–510, London, UK, 1999. Springer-Verlag.

[BFG$^+$99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In *World Congress on Formal Methods (1)*, pages 307–327, 1999.

[BFG$^+$00] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: A validation environment for timed asynchronous systems. In *Computer Aided Verification*, pages 543–547, 2000.

[BG03] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Trans. Comput. Logic*, 4(4):578–651, 2003.

[BG06] Howard Bowman and Rodolfo Gomez. *Concurrency Theory - Calculi and Automata for Modelling Untimed and Timed Concurrent Systems.* Springer-Verlag, 2006.

[BG07] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms: Correction and extension. *ACM Trans. Comput. Log.*, ?(?):1–29, 2007.

[BGM$^+$01] Marius Bozga, Susanne Graf, Laurent Mounier, Iulian Ober, Jean-Luc Roux, and Daniel Vincent. Timed extensions for SDL. In *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 223–240, London, UK, 2001. Springer-Verlag.

[BGS05] Annette Bunker, Ganesh Gopalakrishnan, and Konrad Slind. Live sequence charts applied to hardware requirements specification and verification: A vci bus interface model. *Int. J. Softw. Tools Technol. Transf.*, 7(4):341–350, 2005.

[BH02] 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)*, 2002.

[BH03] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 44, Washington, DC, USA, 2003. IEEE Computer Society.

[BHV00] Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing timed model checking - how the search order matters. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 216–231, London, UK, 2000. Springer-Verlag.

[Bib97] Olivier Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems.* PhD thesis, University of Geneva, 1997.

[BL92] Tommaso Bolognesi and Ferdinando Lucidi. LOTOS-like process algebras with urgent or timed interactions. In *FORTE '91: Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 249–264, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[BLO03] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 256, Washington, DC, USA, 2003. IEEE Computer Society.

[BM07a] Simona Bernardi and Jos&#233; Merseguer. A UML profile for dependability analysis of real-time embedded systems. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 115–124, New York, NY, USA, 2007. ACM Press.

[BM07b] Simona Bernardi and José Merseguer. A uml profile for dependability analysis of real-time embedded systems. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 115–124, New York, NY, USA, 2007. ACM.

[BMN00] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[Bor99] Francis Bordeleau. *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines.* PhD thesis, Carleton University, Ottawa, Canada, 1999.

[BP04] S. Bernardi and D. Petriu. Comparing two UML profiles for non-functional requirement annotations: the spt and qos profiles. In *SVERTS - Satellite Events at the UML Conference*, Lisbon, Portugal, 2004.

[BR95] Egon Börger and Dean Rosenzweig. A mathematical definition of full prolog. *Sci. Comput. Program.*, 24(3):249–286, 1995.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BS98] Egon Börger and Wolfram Schulte. Defining the java virtual machine as platform for provably correct java compilation. In *MFCS '98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 17–35, London, UK, 1998. Springer-Verlag.

[BS00] Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000.

[BS03] E. Börger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[BST98] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling Urgency in Timed Systems. *Lecture Notes in Computer Science*, 1536:103–129, 1998.

[BTF$^+$02] Xiaoying Bai, Wei-Tek Tsai, Ke Feng, Lian Yu, and Ray Paul. Scenario-based modeling and its applications. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 253, Washington, DC, USA, 2002. IEEE Computer Society.

[Buh98] R. J. A. Buhr. Use Case Maps as architectural entities for complex systems. *IEEE Trans. Softw. Eng.*, 24(12):1131–1155, 1998.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[Cas99] Giuseppe Del Castillo. Towards comprehensive tool support for abstract state machines: The ASM workbench tool environment and architecture. In *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pages 311–325, London, UK, 1999. Springer-Verlag.

[CC96] CORPORATE Lockheed Martin Advanced Concepts Center and CORPORATE Rational Software Corporation. *Succeeding with the Booch and OMT methods: a practical approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[CD89] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 428–437, London, UK, 1989. Springer-Verlag.

[CD05] M.L. Crane and J. Dingel. On the semantics of UML state machines: Categorization and comparison. Technical report, School of Computing, Queen's University, Queen's University, Canada, 2005.

[CDH$^+$00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CFJ93] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 450–462, London, UK, 1993. Springer-Verlag.

[CFP01] Flavio Corradini, Gian Luigi Ferrari, and Marco Pistore. On the semantics of durational actions. *Theor. Comput. Sci.*, 269(1-2):47–82, 2001.

[CGL92] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 343–354, New York, NY, USA, 1992. ACM Press.

[CHK05] Pierre Combes, David Harel, and Hillel Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. In *ATVA*, pages 414–428, 2005.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.

[CJ99] S. Casner and V. Jacobson. RFC 2508: Compressing IP/UDP/RTP headers for low-speed serial links, February 1999. Status: PROPOSED STANDARD.

[CK02] María Victoria Cengarle and Alexander Knapp. Towards OCL/RT. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 390–409, London, UK, 2002. Springer-Verlag.

[CL99] Larry L. Constantine and Lucy A. D. Lockwood. *Software for use: a practical guide to the models and methods of usage-centered design.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[CLL97] Chin-Liang Chang, Richard C. Lee, and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, Inc., Orlando, FL, USA, 1997.

[CM99] Brian D. Chance and Bonnie E. Melhart. A taxonomy for scenario use in requirements elicitation and analysis of software systems. *6th Symposium on Engineering of Computer-Based Systems (ECBS'99), 7-12 March 1999, Nashville, TN, USA. IEEE Computer*, 00:232, 1999.

[CNYM99] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering.* Springer, October 1999.

[Coc97] A. Cockburn. Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, Sept-Oct and Nov-Dec, 1997.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[CP04a] C.M.Woodside and D.C. Petriu. Capabilities of the UML profile for schedulability performance and time (spt). In *Workshop SIVOES-SPT held in conjunction with the 10th IEEE RTAS2004*, 2004.

[CP04b] Vittorio Cortellessa and Antonio Pompei. Towards a UML profile for QoS: a contribution in the reliability domain. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 197–206, New York, NY, USA, 2004. ACM Press.

[CR94] J. Chang and D. Richardson. Static and dynamic specification slicing, 1994.

[CSB01] Janette Cardoso and Christophe Sibertin-Blanc. Ordering actions in sequence diagrams of UML. In *23rd International Conference on Information Technology Interfaces, ITI 2001, ISBN 953-96769-3-2, ISSn 1330-1012 , Pula, Croatia, 19/06/01-22/06/01*, pages 3–14, J. Marohnica bb, 10000 Zagreb, Croatia, juin 2001. University Computing Centre Uniersity of Zagreb.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, New York, NY, USA, 1998. ACM Press.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[DAF98] Luigi Logrippo Daniel Amyot, Neil Hart and Pascal Forhan. Formal specification and validation using a scenario-based approach: The GPRS group-call example. In *Selic, B. (Ed.), ObjecTime Workshop on Research in OO Real-Time Modeling, Ottawa, Canada*, 1998.

[DAH04] Ali Echihabi Daniel Amyot and Yong He. Ucmexporter: Supporting scenario transformations from Use Case Maps. In *NOuvelles TEchnnologies de la REpartition, NOTERE 2004, Saidia, Morocco*, pages 134–151, 2004.

[Dav93] Alan M. Davis. *Software requirements: objects, functions, and states.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[DBB97] Benedicte Dano, Henri Briand, and Franck Barbier. An approach based on the concept of use case to produce dynamic object-oriented specifications. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, page 54, Washington, DC, USA, 1997. IEEE Computer Society.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[DFKM98] Jules Desharnais, Marc Frappier, Ridha Khédri, and Ali Mili. Integration of sequential scenarios. *IEEE Trans. Softw. Eng.*, 24(9):695–708, 1998.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[DHHMC95] M. Diefenbruch, E. Heck, J. Hintelmann, and B. Muller-Clostermann. Performance evaluation of SDL systems adjunct by queuing models. In *Proc. of SDL-Forum '95*, London, UK, 1995. Springer-Verlag.

[Die96] Cheryl Dietz. Graphical formalization of real-time requirements. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 366–384, London, UK, 1996. Springer-Verlag.

[DNP99] M. Degermark, B. Nordgren, and S. Pink. RFC 2507: IP header compression, February 1999. Status: PROPOSED STANDARD.

[DP05] R. G. Dromey and Danny Powell. Early requirements defects detection. *TickIT Journal*, 4Q05:3–13, 2005.

[Dro03] R. Geoff Dromey. From requirements to design: Formalizing the key steps. In *1st International Conference on Software Engineering and Formal Methods (SEFM 2003)*, pages 2–, 2003.

[Duf91] David A. Duffy. *Principles of automated theorem proving.* John Wiley & Sons, Inc., New York, NY, USA, 1991.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[EDF96] Earl F. Ecklund, Lois M. L. Delcambre, and Michael J. Freiling. Change cases: use cases that identify future requirements. In *OOPSLA: ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 342–358, San Jose, CA, 1996. ACM Press.

[EDG+05] Huáscar Espinoza, Hubert Dubois, Sébastien Gérard, Julio L. Medina Pasaje, Dorina C. Petriu, and C. Murray Woodside. Annotating UML models with non-functional properties for quantitative analysis. In *MoDELS Satellite Events*, pages 79–90, 2005.

[EE04] EAST-EEA. EAST-EEA. embedded electronic architecture. http://www.east-eea.net, 2004.

[EFH+98] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. RFC 2362: Protocol independent multicast-sparse mode (PIM-SM): Protocol specification, June 1998.

[EGG+01] R. Eschbach, Uwe Glässer, R. Gotzhein, M. von Löwis, and A. Prinz. Formal definition of SDL-2000: Compiling and running SDL specifications as ASM models. *Journal of Universal Computer Science, Special Issue on Abstract State Machines - Theory and Applications*, 2001. Springer-Verlag.

[EH86] E. Allen Emerson and Joseph Y. Halpern. šometimesänd ňot neverřevisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[EM85] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[EMCGP99] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[Esh02] H. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling.* PhD thesis, University of Twente, Enschede, The Netherlands, 2002.

[EW01] R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams – formalising workflow models. Technical report, University of Twente, Department of Computer Science, University of Twente, 2001.

[EW04] R. Eshuis and R. Wieringa. Tool support for verifying UML activity diagrams. *IEEE Trans. Softw. Eng.*, 30(7):437–447, 2004.

[Fen97] W. Fenner. RFC 2236: Internet Group Management Protocol, version 2, November 1997.

[FHD+99] Thomas Firley, Michaela Huhn, Karsten Diethers, Thomas Gehrke, and Ursula Goltz. Timed sequence diagrams and tool-based analysis A case study. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 645–660. Springer, 1999.

[FKG96] J. Froessl, Th. Kropf, and J. Gerlach. An efficient algorithm for real-time symbolic model checking. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 15, Washington, DC, USA, 1996. IEEE Computer Society.

[Fla03] Stephan Flake. Temporal OCL extensions for specification of real-time constraints. In *Workshop Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS'03) at UML'03*, San Francisco, CA, USA, October 2003.

[FM02a] S. Flake and W. Mueller. Specification of real-time properties for UML models. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 277, Washington, DC, USA, 2002. IEEE Computer Society.

[FM02b] Stephan Flake and Wolfgang Mueller. A UML profile for real-time constraints with the OCL. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 179–195, London, UK, 2002. Springer-Verlag.

[FPB87] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[Gar96] H. Garavel. An overview of the eucalyptus toolbox, 1996.

[GBGO00] Nancy D. Griffeth, Ralph Blumenthal, Jean-Charles Grégoire, and Tadashi Ohta. Feature interaction detection contest of the fifth international workshop on feature interactions. *Computer Networks*, 32(4):487–510, 2000.

[GBJ96] J. Rumbaugh G. Booch and I. Jacobson. Unified Modeling Language for Object-Oriented Development(version 0.91 addendum), 1996.

[GBM95] Uwe Glässer, E. Börger, and Wolfgang Müller. Formal definition of an abstract vhdl'93 simulator by ea-machines. In C. Delgado Kloos and Peter T. Breuer, editors, *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[GDO98] V. Grabowski, C. Dietz, and E.-R. Olderog. Semantics for timed Message Sequence Charts via Constraint Diagrams. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC*, Informatik-Bericht Nr. 104, pages 251–260. Humbold-Universitaet zu Berlin/Germany, Juli 1998.

[GGW98] Thomas Gehrke, Ursula Goltz, and Heike Wehrheim. The dynamic models of UML: Towards a semantics and its application in the development process. Technical Report 11/98, Institut für Informatik, Universität Hildesheim, 1998.

[GHS06] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. *Form. Methods Syst. Des.*, 29(2):157–175, 2006.

[GK97] Uwe Glässer and R. Karges. Abstract state machine semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[GK06] Abdelouahed Gherbi and Ferhat Khendek. UML profiles for real-time systems and their applications. *Journal of Object Technology*, 5(4):149–169, 2006.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[GL99] Mark K. Gardner and Jane W.-S. Liu. Analyzing stochastic fixed-priority real-time systems. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 44–58, London, UK, 1999. Springer-Verlag.

[GL04] Gonzalo Génova and Juan Llorens. On the nature of use case-actor relationships. *UPGrade-The European Journal for the Informatics Professional*, V(2):36–42, 2004.

[GL05] Volker Gruhn and Ralf Laue. Specification patterns for time-related properties. In *TIME '05: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 189–191, Washington, DC, USA, 2005. IEEE Computer Society.

[Gli95] Martin Glinz. An integrated formal model of scenarios based on statecharts. In *Proceedings of the 5th European Software Engineering Conference*, pages 254–271, London, UK, 1995. Springer-Verlag.

[GM05] Nicolas Guelfi and Amel Mammar. A formal semantics of timed activity diagrams and its promela translation. In *APSEC'05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 283–290, Washington, DC, USA, 2005. IEEE Computer Society.

[GOO06] Susanne Graf, Ileana Ober, and Iulian Ober. A real-time profile for UML. *Int. J. Softw. Tools Technol. Transf.*, 8(2):113–127, 2006.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1*, pages 438–449, 1993.

[GPFW97] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: a survey. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.

[GR92] Vijay K. Garg and M. T. Ragunath. Concurrent regular expressions and their relationship to petri nets. *Theor. Comput. Sci.*, 96(2):285–304, 1992.

[Gra02] Susanne Graf. Expression of time and duration constraints in SDL. In *3rd SAM Workshop on SDL and MSC, University of Wales Aberystwyth*, number 2599 in LNCS, June 2002.

[GRG93] Peter Graubmann, Ekkart Rudolph, and Jens Grabowski. Towards a Petri Net Based Semantics Definition for Message. In *In: SDL'93 - Using Objects (Editors: O. Faergemand, A. Sarma), North-Holland, October 1993*, October 1993.

[GRS95] Roberto Gorrieri, Marco Roccetti, and Enrico Stancampiano. A theory of processes with durational Actions. *Theoretical Computer Science*, 140(1):73–94, 1995.

[Gua02] Ruoshan Guan. From requirements to scenarios through specifications: A translation procedure from Use Case Maps to LOTOS. Master's thesis, University of Ottawa, Ottawa, Canada, September 2002.

[Gur88] Y. Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current trends in theoretical computer science*, pages 1–57. Computer Science Press, 1988.

[Gur94] Yuri Gurevich. Evolving algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press, 1994.

[Gur99] Y. Gurevich. The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science*, 67:93–124, 1999.

[Gur00] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

[Gur04] Yuri Gurevich. Abstract state machines: An overview of the project. In *FoIKS*, pages 6–13, 2004.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 07(5):11–19, 1990.

[Hal96] Anthony Hall. Using formal methods to develop an atc information system. *IEEE Softw.*, 13(2):66–76, 1996.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HAW03] Yong He, Daniel Amyot, and Alan W. Williams. Synthesizing SDL from Use Case Maps: An experiment. In *SDL 2003: System Design, 11th International SDL Forum, Stuttgart, Germany, July 1-4, 2003. Proceedings*, pages 117–136, 2003.

[HDR04] Jameleddine Hassine, Rachida Dssouli, and Juergen Rilling. Applying reduction techniques to software functional requirement specifications. In *SAM 2004*, pages 138–153, 2004.

[Hey98] S. Heymer. A non–interleaving semantics for msc. In *Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC — SAM'98, Informatik–Berichte Humboldt–Universitt zu Berlin, June 1998.*, 1998.

[Hey00] Stefan Heymer. A semantics for msc based on petri net components. In *SAM 2000*, pages 262–, 2000.

[HGGS00] Tamir Heyman, Daniel Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 20–35, London, UK, 2000. Springer-Verlag.

[HHRS03] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, Transformations and Tools*, pages 1–25, 2003.

[HHRS05] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Stairs towards formal design with sequence diagrams. *Software and System Modeling*, 4(4):355–357, 2005.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 460–463, London, UK, 1997. Springer-Verlag.

[HJRD05] Jameleddine Hassine, Jacqueline Hewitt Juergen Rilling, and Rachida Dssouli. Change impact analysis for requirement evolution using Use Case Maps. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 81–90, Washington, DC, USA, 2005. IEEE Computer Society.

[HJS91] Peter Huber, Kurt Jensen, and Robert M. Shapiro. Hierarchies in coloured petri nets. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 313–341, London, UK, 1991. Springer-Verlag.

[HKL94] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Trans. Softw. Eng.*, 20(1):13–28, 1994.

[HM01] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. Technical Report MSC01-15, The Weizmann Institute of Science, 2001.

[HM02] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched lscs. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, page 193, Washington, DC, USA, 2002. IEEE Computer Society.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, 2003. To Appear.

[HMP91] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *REX Workshop*, pages 226–251, 1991.

[HMP92] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks? In *Automata, Languages and Programming*, pages 545–558, 1992.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[HO03] Zhang Hong Hui and Atsushi Ohnishi. Integration and evolution method of scenarios from different viewpoints. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 183, Washington, DC, USA, 2003. IEEE Computer Society.

[Hoa83] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, 1983.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice/Hall International, April 1985.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill, Inc., New York, NY, USA, 1998.

[HR95] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput.*, 117(2):221–239, 1995.

[HR00] David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical Report MCS00-16, Mathematics & Computer Science, Weizmann Institute Of Science, Jerusalem, Israel, Israel, 2000.

[HRD05a] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. Abstract operational semantics for Use Case Maps. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5*, pages 366–380, 2005.

[HRD05b] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. An ASM Operational Semantics for Use Case Maps. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05), Paris*, pages 467–468. IEEE Computer Society, 2005.

[HRD06] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. Timed Use Case Maps. In *System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 - June 2, 2006, Revised Selected Papers*, pages 99–114, 2006.

[HRD07a] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. Formal Verification of Use Case Maps with Real Time Extensions. In *SDL 2007: Design for Dependable Systems, 13th International SDL Forum, Paris, France, September 18-21, 2007, Proceedings*, pages 225–241, 2007.

[HRD07b] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. Use Case Maps as a Property Specification Language. *Software and System Modeling*, pages –, 2007.

[HSG+94] Pei Hsia, Jayarajan Samuel, Jerry Gao, David Kung, Yasufumi Toyoshima, and Cris Chen. Formal approach to scenario analysis. *IEEE Softw.*, 11(2):33–41, 1994.

[Hug06] James K. Huggins. The ASM michigan webpage. http://www.eecs.umich.edu/gasm/, 2006.

[HW97] Mats P. E. Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 450–467, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[IBM06a] IBM. Eclipse: Eclipse modeling framework (emf). http://www.eclipse.org/emf/, 2006.

[IBM06b] IBM. Graphical editing framework (gef). http://www.eclipse.org/gmf/, 2006.

[ID96a] C. Norris Ip and David L. Dill. State reduction using reversible rules. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 564–567, New York, NY, USA, 1996. ACM.

[ID96b] C. Norris Ip and David L. Dill. Verifying systems with replicated components in murphi. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 147–158, London, UK, 1996. Springer-Verlag.

[Inc04] Cisco Systems Inc. Internet protocol multicast. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ipmulti.htm, 2004. Last accessed, March 2007.

[ISO89] ISO. Information processing systems, osi: Lotos - a formal description technique based on the temporal ordering of observational behaviour., 1989.

[ISO97] ISO. Time extended LOTOS. international standards organization, 1997.

[ISO04] ISO/IEC. Software and system engineering – high-level petri nets – part 1: Concepts, definitions and graphical notation, 2004.

[IT96] ITU-T. Recommendation Z.120. Message Sequence Charts (MSC). Geneva, Switzerland, 1996.

[IT02a] ITU-T. Specification and Description Language, Recommendation Z.100 (SDL). Geneva, Switzerland, 2002.

[IT02b] ITU-T. URN focus group (2002), Recommendation Z.152. UCM: Use Case Maps Notation (UCM). Geneva, Switzerland, 2002.

[IT03a] ITU-T. TTCN, Recommendation Z.140. TTCN: Testing and Test Control Notation version 3 (ttcn-3): Core language, 2003.

[IT03b] ITU-T. URN focus group (2003), Recommendation Z.151. GRL: Goal-oriented Requirement Language (GRL). Geneva, Switzerland, 2003.

[IT04] ITU-T. Recommendation Z.120 (04/04). Message Sequence Charts (MSC). Geneva, Switzerland, 2004.

[ITE04] ITEA. ITEA: EAST-ADL – the EAST-EEA Architecture Description Language. ITEA Project Version 1.02, 2004.

[Jac90] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links, February 1990. Status: PROPOSED STANDARD.

[Jac95] Michael Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[Jac04] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[JEJ94] I. Jacobson, M. Ericsson, and A. Jacobson. *The Object Advantage : Business Process Reengineering With Object Technology (Addison-Wesley Object Technology Series).* Addison-Wesley Professional, September 1994.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Springer, 1992.

[Jen94] Kurt Jensen. An introduction to the theoretical aspects of coloured petri nets. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 230–272, London, UK, 1994. Springer-Verlag.

[Jia05] Bo Jiang. Combining graphical scenarios with a requirements management system. Master's thesis, SITE, University of Ottawa, Canada, 2005.

[JP86] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[JP01] Bengt Jonsson and Gerardo Padilla. An execution semantics for MSC-2000. In *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 365–378, London, UK, 2001. Springer-Verlag.

[JR94] Daniel Jackson and Eugene J. Rollins. Chopping: A generalization of slicing. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[JS04] S. Jansamak and A. Surarerks. Formalization of UML statechart models using concurrent regular expressions. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 83–88, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[jUC06] jUCMNav. jUCMNav Project (tool, documentation, and meta-model). http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/WebHome, 2006. Last accessed, October 2007.

[jUC07] jUCMNav. UCM component of URN meta-model. http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/URNMetaModel, 2007. Last accessed, Feb 2008.

[JW02] David N. Jansen and Roel Wieringa. Extending ctl with actions and real time. *J. Log. Comput.*, 12(4):607–621, 2002.

[KA07] Jason Kealey and Daniel Amyot. Enhanced use case map traversal semantics. In *SDL 2007: Design for Dependable Systems, 13th International SDL Forum, Paris, France, September 18-21, 2007, Proceedings*, pages 133–149, 2007.

[KC98] K. Khordoc and E. Cerny. Semantics and verification of action diagrams with linear timing. *ACM Trans. Des. Autom. Electron. Syst.*, 3(1):21–50, 1998.

[KC05a] Sascha Konrad and Betty H. C. Cheng. Facilitating the construction of specification pattern-based properties. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 329–338, Washington, DC, USA, 2005. IEEE Computer Society.

[KC05b] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 372–381, 2005.

[KC06] Tai Hyo Kim and Sung Deok Cha. Timed high-level Message Sequence Charts for Real-Time System Design. In *System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 - June 2, 2006, Revised Selected Papers*, pages 82–98, 2006.

[KCH01] Saehwa Kim, Sukjae Cho, and Seongsoo Hong. Automatic implementation of real-time object-oriented models and schedulability issues. In *WORDS '01: Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, page 137, Washington, DC, USA, 2001. IEEE Computer Society.

[KCH05] J. Klein, B. Caillaud, and L. Hlout. Merging scenarios. In *Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems, FMICS'04*, volume 133 of *Electronic Notes in Theoretical Computer Science*, pages 193–215, Linz, Austria, 2005.

[Kho96] K. Khordoc. *Action diagrams: A methodology for the specification and verification of real-time systems.* PhD thesis, McGill University, Montreal, Canada, 1996.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[KL98] J.-P. Katoen and L. Lambert. Pomsets for Message Sequence Charts. In H. König and P. Langendörfer, editors, *Formale Beschreibungstechniken für verteilte Systeme*, pages 197–207, Cottbus, Germany, June 1998. GI/ITG, Shaker Verlag.

[KM94a] B. Knight and J. Ma. Time representation: A taxonomy of temporal models. *AI Review*, 7:401–419, 1994.

[KM94b] Kai Koskimies and Erkki Makinen. Automatic synthesis of state machines from trace diagrams. *Software - Practice and Experience*, 24(7):643–658, 1994.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.

[KP91] Yonit Kesten and Amir Pnueli. Timed and hybrid statecharts and their textual representation. In *Proceedings of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 591–620, London, UK, 1991. Springer-Verlag.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[KSTV03] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state-based models. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 34, Washington, DC, USA, 2003. IEEE Computer Society.

[KW01] Jochen Klose and Hartmut Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–527, London, UK, 2001. Springer-Verlag.

[Lak94] C. A. Lakos. Object petri nets - definition and relationship to coloured nets. tr94-3. Technical report, Computer Science Department, University of Tasmania, 1994.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam00] Axel Van Lamsweerde. Requirements engineering in the year 00: a research perspective. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 5–19, New York, NY, USA, 2000. ACM Press.

[Lar01] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[LDD06] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12, New York, NY, USA, 2006. ACM Press.

[LFHH91] Luigi Logrippo, Mohammed Faci, and Mazen Haj-Hussein. An introduction to lotos: Learning by examples. *Computer Networks and ISDN Systems*, 23(5):325–342, 1991.

[LK91] C. Lakos and C. Keen. Loopn++: A new language for object-oriented petri nets. In *Proc. Modelling and Simulation*, pages 369–374, 1991.

[LKR07] Grunske Lars, Winter Kirsten, and Colvin Robert. Timed behavior trees and their application to verifying real-time systems. In *Proceedings of 18th Australian Conference on Software Engineering (ASWEC 2007)*, April 2007. accepted for publication.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[LL93a] Peter B. Ladkin and Stefan Leue. What do Message Sequence Charts mean? In *FORTE 93*, pages 301–316, 1993.

[LL93b] Luc Leonard and Guy Leduc. An enhanced version of timed LOTOS and its application to a case study. In *FORTE*, pages 483–498, 1993.

[LL99a] Xuandong Li and Johan Lilius. Timing Analysis of Message Sequence Charts. Technical Report TUCS-TR-255, TUCS - Turku Centre for Computer Science, 24, 1999.

[LL99b] Xuandong Li and Johan Lilius. Timing analysis of UML sequence diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 661–674. Springer, 1999.

[LMH00] P. Le Maigat and L. Hélouët. A (max,+) approach for time in message sequence charts. In R. Boel and G. Stremersch, editors, *Proceedings of the 5th Workshop on Discrete Event Systems*, pages 83–92, Ghent, Belgium, 2000. Kluwer Academic Publishers.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[LMM05] Luigi Lavazza, Sandro Morasca, and Angelo Morzenti. A dual language approach to the development of time-critical systems. *Electr. Notes Theor. Comput. Sci.*, 116:227–239, 2005.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, New York, NY, USA, 1985. ACM Press.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LR01] M. M. Lehman and J. F. Ramil. Evolution in software and related areas. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 1–16, New York, NY, USA, 2001. ACM Press.

[LRG04] Hon F. Li, Juergen Rilling, and Dhrubajyoti Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. *Automated Software Engg.*, 11(1):63–89, 2004.

[LS98] Mikael Lindvall and Kristian Sandahl. How well do experienced software developers predict software change? *The Journal of Systems and Software*, 43(1):19–27, 1998.

[LS02] Hung Ledang and Jeanine Souquieres. Contributions for modelling UML state-charts in b. In *IFM*, pages 109–127, 2002.

[Luc01] Andrea De Lucia. Program slicing: Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, November 2001.

[MAB$^+$01] Andrew Miga, Daniel Amyot, Francis Bordeleau, Donald Cameron, and C. Murray Woodside. Deriving Message Sequence Charts from Use Case Maps scenario specifications. In *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 268–287, London, UK, 2001. Springer-Verlag.

[Mai98] N. A. M. Maiden. CREWS-SAVRE: Scenarios for acquiring and validating requirements. *Automated Software Engineering: An International Journal*, 5(4):419–446, October 1998.

[Mau96] S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, 1996.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[MdBFS04] Kyas Marcel and de Boer Frank S. On message specification in OCL. In Frank S. de Boer and Marcello Bonsangue, editors, *Compositional Verification in UML*, volume 101 of *entcs*, pages 73–93. elsevier, 2004.

[Mer74] Philip Meir Merlin. *A study of the recoverability of computing systems.* PhD thesis, University of California, Irvine, 1974.

[Mes90] David G. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, 1990.

[MHK02] Rami Marelly, David Harel, and Hillel Kugler. Multiple instances and symbolic variables in executable sequence charts. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–100, New York, NY, USA, 2002. ACM Press.

[Mig98] Andrew Miga. Application of Use Case Maps to system design with tool support. Master's thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MLLG01] Grant Martin, Luciano Lavagno, and Jean Louis-Guerin. Embedded UML: a merger of real-time UML and co-design. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 23–28, New York, NY, USA, 2001. ACM Press.

[Mos90] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 575–631. Elsevier, Amsterdam, 1990.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MP96] Z Manna and Amir Pnueli. Clocked transition systems. Technical report, Stanford University, Stanford, CA, USA, 1996.

[MPW92] Robin Milner, Joachim Parrow, and D. Walker. A calculus of mobile processes (parts i and ii). *Information and Computation*, 100:1–77, 1992.

[MR94] S. Mauw and M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–278, 1994.

[MR97] S. Mauw and M. A. Reniers. High-level message sequence charts. In *Proceedings of the Eighth SDL Forum (SDL'97)*, pages 291–306, 1997.

[MR99] S. Mauw and M. A. Reniers. Operational Semantics for MSC'96. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(17):1785–1799, 1999.

[MRK+97] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology*, 6(1):31–79, 1997.

[MS93] N. Meng-Siew. Reasoning with timing constraints in message sequence charts. Master's thesis, University of Stirling, Scotland, U.K., August 1993.

[MSP96] Andrea Maggiolo-Schettini and Adriano Peron. Retiming techniques for statecharts. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 55–71, London, UK, 1996. Springer-Verlag.

[MT98] L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking, 1998.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[NB03] Muan Yong Ng and Michael Butler. Towards formalizing UML state diagrams in CSP. *sefm*, 00:138, 2003.

[NE00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[NFGR92] Rocco De Nicola, Alessandro Fantechi, Stefania Gnesi, and Gioia Ristori. An action based framework for verifying logical and behavioural properties of concurrent systems. In *CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 37–47, London, UK, 1992. Springer-Verlag.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[NS92] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 526–548, London, UK, 1992. Springer-Verlag.

[NS94] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, atp: theory and application. *Inf. Comput.*, 114(1):131–178, 1994.

[NV90] Rocco De Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Proceedings of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes*, pages 407–419, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[OGO04] I. Ober, S. Graf, and I. Ober. Model checking of UML models via a mapping to communicating extended timed automata, 2004.

[OGO06] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *Int. J. Softw. Tools Technol. Transf.*, 8(2):128–145, 2006.

[OK01] Iulian Ober and Alain Kerbrat. Verification of quantitative temporal properties of SDL specifications. In *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 182–202, London, UK, 2001. Springer-Verlag.

[OME07] OMEGA. OMEGA consortium. webpage of the omega ist project. http://www-omega.imag.fr/, 2007. Last accessed, March 2007.

[OMG02] OMG. Response to the OMG RFP for Schedulability, Performance and Time, v. 2.0. OMG document ad/2002-03-04, March 2002.

[OMG03] OMG. Unified Modeling Language specification, 2003. Version 1.5, March 2003 via `http://www.omg.org`.

[OMG05] OMG. UML 2.0 superstructure specification, oct 2005.

[OMG06] OMG. Object management group. UML profile for modeling quality of service and fault tolerant characteristics and mechanisms. OMG document formal/06-05-02, May 2006.

[OO90] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. Softw. Eng.*, 16(3):268–280, 1990.

[ÖP98] Gunnar Övergaard and Karin Palmkvist. A Formal Approach to Use cases and their Relationships. In P.-A. Muller and J. Bézivin, editors, *Proceedings of ≪UML≫'98: Beyond the Notation*, pages 309–317. Ecole Supérieure des Sciences Appliquées pour l'Ingénieur – Mulhouse, Université de Haut-Alsace, France, 1998.

[OSR95] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, 1995.

[Pal99] Ivan P Paltor. The semantics of UML state machines. Technical report, Turku Centre for Computer Science, 1999.

[PAWJ03] Dorin Bogdan Petriu, Daniel Amyot, C. Murray Woodside, and Bo Jiang. Traceability and evaluation in scenario analysis by use case maps. In *Scenarios: Models, Transformations and Tools*, pages 134–151, 2003.

[PB00] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.

[PC90] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, 1990.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[Pet77] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, 1977.

[Pfl01] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[PMI94] David Lorge Parnas, Jan Madey, and Michal Iglewski. Precise documentation of well-structured programs. *IEEE Trans. Softw. Eng.*, 20(12):948–976, 1994.

[PMS94] Adriano Peron and Andrea Maggiolo-Schettini. Transitions as interrupts: A new semantics for timed statecharts. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 806–821, London, UK, 1994. Springer-Verlag.

[PN91] Santiago Pavón and Martín Llamas Nistal. The testing functionalities of lola. In *FORTE '90: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 559–562, Amsterdam, The Netherlands, The Netherlands, 1991. North-Holland Publishing Co.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE, IEEE Computer Society Press.

[Poh96] Klaus Pohl. Requirements Engineering: An Overview. Technical Report AIB-05-1996, RWTH Aachen, 1996.

[PW02] Dorin C. Petriu and C. Murray Woodside. Software performance models from system scenarios in Use Case Maps. In *TOOLS '02: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 141–158, London, UK, 2002. Springer-Verlag.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RA98] Colette Rolland and Camille B. Achour. Guiding the construction of textual use case specifications. *Data Knowledge Engineering*, 25(1-2):125–160, 1998.

[RAC+98] C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyté;, A. Sutcliffe, N. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois, and P. Heymans. A proposal for a scenario classification framework. *Requir. Eng.*, 3(1):23–47, 1998.

[Ram74] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[RBL+90] James R. Rumbaugh, Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, October 1990.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[RK97] Jürgen Ruf and Thomas Kropf. Symbolic model checking for a discrete clocked temporal logic with intervals. In *Proceedings of the IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, pages 146–163, London, UK, UK, 1997. Chapman & Hall, Ltd.

[RKA06] Jean-François Roy, Jason Kealey, and Daniel Amyot. Towards integrated tool support for the user requirements notation. In *SAM 2006*, pages 198–215, 2006.

[RL07a] RT-LOTOS. CADP. construction and analysis of distributed processes. http://www.inrialpes.fr/vasy/cadp/, 2007. Last accessed, March 2007.

[RL07b] RT-LOTOS. RT-LOTOS. software and tools for communicating systems. http://www.laas.fr/RT-LOTOS/, 2007. Last accessed, March 2007.

[RM99] Sita Ramakrishnan and John McGregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22, 1999*, 1999.

[RMSM+96] Y. S. Ramakrishna, P. M. Melliar-Smith, L. E. Moser, L. K. Dillon, and G. Kutty. Interval logics and their decision procedures: part i: an interval logic. *Theor. Comput. Sci.*, 166(1-2):1–47, 1996.

[RP96] C. Rolland and N. Prakash. A Proposal for Context-Specific Method Engineering. In S. Brinkkemper, K. Lyytinen, and R. J. Welke, editors, *Proceedings of the IFIP TC8 WG8.1/8.2 Working Conference on Method Engineering*, pages 191–208, Atlanta, Georgia, 1996. Chapman & Hall.

[RR88] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988.

[RR95] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52, New York, NY, USA, 1995. ACM Press.

[RS97] Jean-François Raskin and Pierre-Yves Schobbens. State clock logic: A decidable real-time logic. In *HART '97: Proceedings of the International Workshop on Hybrid and Real-Time Systems*, pages 33–47, London, UK, 1997. Springer-Verlag.

[RT01] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM Press.

[RvKWR01] E.E. Roubtsova, J. van Katwijk, W.J.Toetenel, and R.C.M.de Rooij. Real-time systems: Specification of properties in UML. In *ASCI 2001 conference, Het Heijderbos, Heijen, The Netherlands, May 30 - June 1 2001*, pages 188–195, 2001.

[SAn07] SAnToS Laboratory. Spec patterns. http://patterns.projects.cis.ksu.edu/documentation/patterns/ctl. 2007. Last accessed, July 2007.

[Sch06] J. Schmid. Executing ASM specifications with AsmGofer. http://www.tydo.de/AsmGofer, 2006.

[SCHK+04] Raffaella Settimi, Jane Cleland-Huang, Oussama Ben Khadra, Jigar Mody, Wiktor Lukasik, and Chris DePalma. Supporting software evolution through dynamically retrieving traces to UML artifacts. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop on (IWPSE'04)*, pages 49–54, Washington, DC, USA, 2004. IEEE Computer Society.

[SD05] Jun Sun and Jin Song Dong. Model checking live sequence charts. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 529–538, Washington, DC, USA, 2005. IEEE Computer Society.

[SDV95] Stephane Somé, Rachida Dssouli, and Jean Vaucher. From scenarios to timed automata: Building specifications from users requirements. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 48, Washington, DC, USA, 1995. IEEE Computer Society.

[SDV96] Stephane Somé, Rachida Dssouli, and Jean Vaucher. Toward an automation of requirements engineering using scenarios. *Journal of Computing and Information*, 2(1):1110–1132, 1996.

[SE94] S.V. Campos and E. Clarke. Real-Time Symbolic Model Checking for Discrete Time Models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Develpment.* World Scientific Press, AMAST Series in Computing, 1994.

[SFR97] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 240, Washington, DC, USA, 1997. IEEE Computer Society.

[SG91] Gil Shurek and Orna Grumberg. The modular framework of computer-aided verification. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 214–223, London, UK, 1991. Springer-Verlag.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

[SGY$^+$04] Wuwei Shen, Mohsen Guizani, Zijiang Yang, Kevin J. Compton, and James Huggins. Execution of a requirement model in software development. In *IASSE*, pages 203–208, 2004.

[SH96] Anthony M. Sloane and Jason Holdsworth. Beyond traditional program slicing. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 180–186, New York, NY, USA, 1996. ACM Press.

[SH05] Harald Störrle and Jan Hendrik Hausmann. Towards a formal semantics of UML 2.0 activities. In *Software Engineering*, pages 117–128, 2005.

[SHE01] Margaret H. Smith, Gerard J. Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 14–22, Washington, DC, USA, 2001. IEEE Computer Society.

[SHR07] Maryam Shiri, Jameleddine Hassine, and Juergen Rilling. Modification analysis support at the requirements level. In *IWPSE '07: Proceedings of the International Workshop on Principles of Software Evolution*, Washington, DC, USA, 2007. IEEE Computer Society.

[Sin04] Richard O. Sinnott. The formal, tool supported development of real time systems. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, pages 388–395, Washington, DC, USA, 2004. IEEE Computer Society.

[SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[SKW00] M. Saksena, P. Karvelas, and Y. Wang. Automatic synthesis of multi-tasking implementations from real-time object-oriented models. In *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 360, Washington, DC, USA, 2000. IEEE Computer Society.

[SL03] Wuwei Shen and Shaoying Liu. Formalization, testing and execution of a use case diagram. In *ICFEM*, pages 68–85, 2003.

[Som04] Stephane S. Some. An environment for use cases based requirements engineering. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04)*, pages 364–365, Washington, DC, USA, 2004. IEEE Computer Society.

[Som06] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Spe06] SpecExplorer. Microsoft research: Spec explorer tool. http://research.microsoft.com/specexplorer/, 2006.

[SS96] M. R. Strens and R. C. Sugden. Change analysis: A step towards meeting the challenge of changing requirements. In *ECBS '96: Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[SS97] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A Good Practice Guide.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[SS00] S. Sendall and A. Strohmeier. From use cases to system operation specification. In *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939, pages 1–15, London, UK, 2000. Springer-Verlag.

[SS01] Shane Sendall and Alfred Strohmeier. Specifying concurrent system behavior and timing constraints using ocl and UML. In *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, pages 391–405, London, UK, 2001. Springer-Verlag.

[Sto04a] Harald Storrle. Semantics of control-flow in UML 2.0 activities. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 235–242, Washington, DC, USA, 2004. IEEE Computer Society.

[Stö04b] Harald Störrle. Structured nodes in UML 2.0 activities. *Nordic J. of Computing*, 11(3):279–302, 2004.

[Stö05] Harald Störrle. Semantics and verification of data flow in UML 2.0 activities. *Electr. Notes Theor. Comput. Sci.*, 127(4):35–52, 2005.

[SW98] Judith A. Stafford and Alexander L. Wolf. Architecture-level dependence analysis in support of software maintenance. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 129–132, New York, NY, USA, 1998. ACM Press.

[SZ02] Emil Sekerinski and Rafik Zurob. Translating statecharts to b. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 128–144, London, UK, 2002. Springer-Verlag.

[TBW94] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Syst.*, 6(2):133–151, 1994.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[TSBC05] Omar Tahir, Christophe Sibertin-Blanc, and Janette Cardoso. A causality-based semantics for UML sequence diagrams. In *IASTED Conf. on Software Engineering*, pages 106–111, 2005.

[TTo07] TTool. TTool. a toolkit for editing and validating turtle diagrams. http://labsoc.comelec.enst.fr/turtle/, 2007. Last accessed, March 2007.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[TUR00] K. TURNER. Formalising the chisel feature notation. In *Proc. 6th. Feature Interactions in Telecommunications and Software Systems (ed. by CALDER M. H., MA- GILL E. H.)*, pages 241–256. IOS Press, Amsterdam, Netherlands, 2000.

[TYZP05] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul. Rapid embedded system testing using verification patterns. *IEEE Softw.*, 22(4):68–75, 2005.

[UKM03] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003.

[Val91] Antti Valmari. A stubborn attack on state explosion. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165, London, UK, 1991. Springer-Verlag.

[Var91] Moshe Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1-2):79–98, 1991.

[vdB94] Michael von der Beeck. A comparison of statecharts variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.

[vdB01] Michael von der Beeck. Formalization of UML-statecharts. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 406–421, London, UK, 2001. Springer-Verlag.

[vK01] Antje von Knethen. A trace model for system requirements changes on embedded systems. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 17–26, New York, NY, USA, 2001. ACM Press.

[VK05] Valdis Vitolins and Audris Kalnins. Semantics of UML 2.0 activity diagram for business modeling by means of virtual machine. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, pages 181–194, Washington, DC, USA, 2005. IEEE Computer Society.

[Wal95] Charles Wallace. The semantics of the C++ programming language. In *Specification and validation methods*, pages 131–164. Oxford University Press, Inc., New York, NY, USA, 1995.

[WD04] Lian. Wen and R. Geoff. Dromey. From requirements change to design change: A formal path. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, pages 104–113, Washington, DC, USA, 2004. IEEE Computer Society.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[WH02] Farn Wang and Pao-Ann Hsiung. Efficient and user-friendly verification. *IEEE Trans. Comput.*, 51(1):61–83, 2002.

[Win04] Kirsten Winter. Formalising behaviour trees with csp. In *IFM*, pages 148–167, 2004.

[WWH05] Farn Wang, Rong-Shiung Wu, and Geng-Dian Huang. Verifying timed and linear hybrid rule-systems with red. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), Taipei, Taiwan, Republic of China*, pages 448–454, 2005.

[XMY+01] Li Xuandong, Cui Meng, Pei Yu, Zhao Jianhua, and Zheng Guoliang. Timing analysis of UML activity diagrams. In *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, pages 62–75, London, UK, 2001. Springer-Verlag.

[Yi00] Z. Yi. CNAP specification and validation: A design methodology using Lotos and UCM. Master's thesis, SITE, University of Ottawa, Canada, 2000.

[YLWD05] W. L. Yeung, Karl R. P. H. Leung, Ji Wang, and Wei Dong. Improvements towards formalizing UML state diagrams in CSP. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 176–184, Washington, DC, USA, 2005. IEEE Computer Society.

[Yov97] Serjio Yovine. KRONOS: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), December 1997.

[YsZ03] Dong Yang and Shen sheng Zhang. Using π- calculus to formalize UML activity diagram. In *10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003), 7-10 April 2003, Huntsville, AL, USA*, pages 47–54, 2003.

[Zen05] Yong Xiang Zeng. Transforming Use Case Maps to the core scenario model representation. Master's thesis, SITE, University of Ottawa, Canada, 2005.

[Zha98] Jianjun Zhao. Applying slicing technique to software architectures. In *Proceedings of 4th IEEE International Conference on Engineering of Complex Computer Systems*, pages 87–98, August 1998.

[ZHJ04] Tewfic Ziadi, Loic Helouet, and Jean-Marc Jezequel. Revisiting statechart synthesis with an algebraic approach. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 242–251, Washington, DC, USA, 2004. IEEE Computer Society.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

[ZK02] Tong Zheng and Ferhat Khendek. An extension for MSC-2000 and its application. In *Telecommunications and beyond: The Broader Applicability of SDL and MSC, Third International Workshop, SAM 2002, Aberystwyth, UK, June 24-26. Revised Papers*, pages 221–232, 2002.

[ZKH02] Tong Zheng, Ferhat Khendek, and Loc Hélouët. A semantics for timed MSC. *Electr. Notes Theor. Comput. Sci.*, 65(7), 2002.