Animation-based Validation of Reactive Software Systems using Behavioural Models

Óscar Rafael da Silva Ferreira Ribeiro

UMinho | 2009

Universidade do Minho
Escola de Engenharia

Óscar Rafael da Silva Ferreira Ribeiro

Animation-based Validation of Reactive Software Systems using Behavioural Models

Novembro de 2009

Universidade do Minho
Escola de Engenharia

Óscar Rafael da Silva Ferreira Ribeiro

Animation-based Validation of Reactive Software Systems using Behavioural Models

Doutoramento em Informática

Trabalho efectuado sob a orientação do
Doutor João Miguel Lobo Fernandes

Novembro de 2009

Abstract

During the development of software systems, validation is a crucial activity to guarantee that the software system fulfills the users' needs and expectations. A key issue to have a successful validation consists in adopting a process where users and clients can actively discuss the requirements of the system under development.

A reactive system is expected to continuously interact with its environment. Usually, the interaction of a reactive system with its environment is supported by a set of nonterminating processes that operate in parallel. During the interaction, the reactive system must answer to high-priority events, even when the system is executing something else. Due to above characteristics, the behaviour of reactive systems can be very complex.

The approach suggested in this thesis assumes that the requirements of reactive software systems are partially described by use case diagrams, and each use case is detailed by a collection of scenario descriptions. Within this approach, one can obtain, from a set of behavioural scenarios of a given system, an executable behavioural model that can support, when complemented with animation- and domain-specific elements, a graphical animation for reproducing that set of scenarios for validation purposes. Animating the scenarios using graphical elements from the application domain ensures an effective involvement of the users in the system's validation.

The Coloured Petri nets (CPNs) modelling language is used as the notation to obtain the behavioural models, due to its natural support for mechanisms like concurrency, synchronisation, and resource sharing and its tool support. The obtained CPN model is guaranteed to be (1) parametric, allowing an easy modification of the initial conditions of the scenarios, (2) environment-descriptive, meaning that it includes the state of the relevant elements of the environment, and (3) animation-separated, implying that the elements related to the animation are separated from the other ones.

We validate our approach based on its application to three case studies of reactive systems.

Resumo

Durante o desenvolvimento de sistemas de software, a validação é uma actividade crucial para garantir que o sistema de software satisfaz as necessidades e expectativas do utilizador. O sucesso na validação consiste na utilização de um processo onde os utilizadores e os clientes possam discutir de uma forma activa os requisitos do sistema que está a ser desenvolvido.

Um sistema reactivo está continuamente em interacção com o seu ambiente, que é geralmente suportada por um conjunto de processos intermináveis que operam em paralelo. Durante a interacção, o sistema reactivo deverá responder aos eventos com alta prioridade, mesmo quando o sistema está a executar algo diferente. Devido às características anteriores, o comportamento dos sistemas reactivos pode ser muito complexo.

A abordagem sugerida nesta tese assume que os requisitos de sistemas reactivos são em parte descritos por diagramas de casos de uso e que cada caso de uso é detalhado por uma colecção de descrições de cenários. Nesta abordagem, é possível obter, a partir de um conjunto de cenários de um dado sistema, um modelo comportamental que seja executável e que suporte, quando complementado com elementos específicos, uma animação gráfica que reproduza aquele conjunto de cenários para fins de validação. A animação dos cenários utilizando elementos gráficos do domínio da aplicação garante um envolvimento efectivo dos utilizadores na validação do sistema.

A linguagem de modelação redes de Petri coloridas (CPNs) é usada como a notação para obter os modelos comportamentais, devido ao seu suporte natural a mecanismos como a concorrência, sincronização e partilha de recursos, e às suas ferramentas de suporte. Se as recomendações da abordagem proposta foram seguidas, temos a garantia que o modelo CPN: (1) parametriza as condições iniciais dos cenários, (2) contém uma descrição do ambiente, incluindo o estado dos seus elementos, e (3) separa os elementos relacionados com a animação dos outros elementos do modelo.

A validação da nossa abordagem tem por base a sua aplicação a três casos de estudo de sistemas reactivos.

Acknowledgments

First of all, I would like to thank my supervisor, João Miguel Fernandes, for creating the stimulating environment for doing the research for this dissertation. He encouraged and guided me to explore the directions for research. Afterwards, he helped me to shape the research presented in this thesis.

This work was supported by Fundação para a Ciência e Tecnologia (FCT), under the grant with reference SFRH/BD/19718/2004.

I wish to express my gratitude to my parents and my brother for their understanding, love and continuous support throughout these years.

Finally, I would like to thank my wife, Natália for her continuous encouragement, support, patience and love during the many months that I needed to finish this thesis.

Contents

Abstract i

Resumo iii

Acknowledgments v

List of Figures xi

I Background 1

1 Introduction 3
- 1.1 Reactive Software Systems . . . 4
- 1.2 Motivation . . . 6
- 1.3 Problem Statement . . . 7
- 1.4 Aims . . . 8
- 1.5 Approach Taken . . . 9
- 1.6 Contribution . . . 11
- 1.7 Overview . . . 12

2 Behavioural Models 15
- 2.1 Use cases . . . 16
- 2.2 Interactions . . . 17
  - 2.2.1 Introduction to interactions . . . 17
  - 2.2.2 Sequence diagrams . . . 18
  - 2.2.3 Metamodel for UML2 sequence diagrams . . . 20
- 2.3 Coloured Petri Nets . . . 25

2.3.1 Basic concepts 26
2.3.2 Creating a CPN model 29
2.3.3 Tool support 35

3 Software Requirements **37**
3.1 Introduction 38
3.2 Requirements Engineering Process 41
3.3 Requirements Elicitation and Analysis 43
3.3.1 General considerations 43
3.3.2 Techniques and approaches for elicitation 46
3.4 Requirements Validation 50

II Contribution **57**

4 Transforming Sequence Diagrams into a CPN Model **59**
4.1 Plain Sequence Diagrams 60
4.2 Sequence Diagrams with High-level Operators 62
4.2.1 Alternative choice 62
4.2.2 Optional 63
4.2.3 Parallel composition 63
4.2.4 Weak sequencing 65
4.2.5 Looping 66
4.3 Tool Support 67
4.3.1 Description of the involved metamodels 68
4.3.2 Details of the transformation 72

5 Enriching CPN models for Animation **77**
5.1 Introduction 78
5.2 Mapping sequence diagrams into a CPN model 78
5.3 Data representation for the environment 81
5.4 Animation of messages in the sequence diagrams 83
5.5 Initial conditions for scenario execution 84
5.6 Building an animation 86
5.6.1 Initial considerations 86
5.6.2 Static part of the animation 88

5.6.3 Dynamic part of the animation 88
5.6.4 Scripting language 90

6 Case Studies **93**
6.1 Reactor System 94
6.1.1 General description 94
6.1.2 A shobi-PN based CPN model 95
6.1.3 A scenario-based CPN model 101
6.1.4 Building an animation 105
6.1.5 A SIP-net approach to model the reactor 106
6.2 Elevator Controller System 125
6.2.1 General description 125
6.2.2 Use cases descriptions 127
6.2.3 Expressing scenarios by a CPN model 134
6.2.4 Building an animation 135
6.3 Check-in System 141
6.3.1 General description 141
6.3.2 Expressing scenarios by a CPN model 144
6.3.3 Building an animation 152
6.4 Discussion 153

7 Related Work **155**
7.1 Formalization of Use Case Descriptions 156
7.2 Generation of State-based Models from Scenarios 157
7.3 Synthesis of Petri Net Models from Scenarios 159
7.4 Animating Formal Specifications of Requirements 166

8 Conclusions and Future Work **169**
8.1 Contributions 170
8.2 Future Work 171

Bibliography **173**

List of Figures

1.1 Process for the approach. 10

2.1 An example of a use case diagram. 17
2.2 An example of a sequence diagram. 19
2.3 Part of UML2 metamodel for Interactions. 21
2.4 Part of UML2 metamodel for Lifelines. 22
2.5 Part of UML2 metamodel for Messages. 23
2.6 Part of UML2 metamodel for Occurrence Specifications. . . . 24
2.7 Part of UML2 metamodel for Combined Fragments. 25
2.8 A small CPN model. 26
2.9 CPN-ML code used in the example of CPN model 27
2.10 Sketch of the options that passengers at each floor have to call an elevator car. 30
2.11 CPN model that for the example, that uses only the colour set UNIT. 31
2.12 Example of CPN model for the example. 32
2.13 CPN-ML code used in the CPN model in Figure 2.12. 33
2.14 Example of CPN model. 34
2.15 CPN-ML code used in the CPN model in Figure 2.14. 34

3.1 The topics for the software requirements. 40
3.2 Spiral model for requirements engineering process. 42

4.1 Transforming a sequence diagram with only one message. . . 60
4.2 Example of transform a sequence diagram without high-level operators 61
4.3 Example with the alternative choice operator (*alt*) 63

4.4 The option operator (*opt*) expressed by an alternative choice. 64
4.5 Example with the parallel composition operator (*par*) 64
4.6 Example with the weak sequencing operator (*seq*) 65
4.7 Another example with the weak sequencing operator (*seq*) . . 66
4.8 Example with the looping operator (*loop*). 67
4.9 An example of UML2 sequence diagram. 68
4.10 The elements of a sequence diagram. 69
4.11 XMI code. 70
4.12 Metamodel of CPN modelling language. 71
4.13 The XMI code representation of a CPN model 72
4.14 An overview of the transformation. 73
4.15 The ATL transformation rule to be applied to a message occurrence specification. 73
4.16 The ATL transformation rule to be applied to a message. . . 74
4.17 The ATL transformation rule to be applied to a lifeline. . . . 75
4.18 The ATL transformation rule to be applied to an interaction. 76

5.1 Sequence Diagram describing the "Service Floor" use case. . . 79
5.2 CPN model representing a sequence diagram for UC2 "Service Floor". 80
5.3 CPN model for the execution of the message lightDirInd. . . . 84
5.4 Subpage of CPN model to capture events from the SceneBeans animation. 85
5.5 CPN module to initialise the environment values and the SceneBeans animation. 86
5.6 Three representations for the light direction indicator. 89

6.1 The environment of industrial reactor system. 94
6.2 A CPN model for reactor system. 98
6.3 CPN-ML code for colour set StorageVessel 99
6.4 A CPN module for toggle switch. 99
6.5 CPN-ML code for colour set Switch. 99
6.6 A CPN module for filling a measuring vessel. 100
6.7 A CPN module for emptying measuring vessel. 100
6.8 A sequence diagram describing some scenarios of using the reactor system 101

6.9 A sequence diagram describing the behaviour of vessels. . . . 102
6.10 A sequence diagram describing the preparation of car. 102
6.11 CPN from the sequence diagram presented in Figure 6.8. . . . 103
6.12 CPN to represent the behaviour of vessels (see Figure 6.9) . . 104
6.13 A screenshot of the animation of the reactor system 105
6.14 Design flow of the approach. 108
6.15 An example of an SIP-net model. 114
6.16 Definitions and declarations in PROMELA. 115
6.17 PROMELA process for specifying the SIP-net model example. 116
6.18 Definition of the enabled conditions. 118
6.19 Truth values of guards. 119
6.20 Haskell data types to represent SIP-net models. 121
6.21 Haskell specification of the SIP-net model example. 122
6.22 Context diagram for the elevator controller. 126
6.23 Use case diagram for the elevator controller system. 128
6.24 Sequence Diagram describing the "Service Floor" use case. . . 129
6.25 Sequence Diagram describing the main scenario of the use case UC2 "Service Floor", and some of its variations. 132
6.26 Sequence Diagram describing the main scenario of the use case UC1 "Travel to Floor". 133
6.27 CPN model representing a sequence diagram for UC1. 134
6.28 CPN model representing a sequence diagram for UC1. 135
6.29 Diagram that associates graphical representations to some of the entities of the environment of the elevator controller. . . . 137
6.30 A screenshot of an animation for the elevator controller system. 138
6.31 Sequence diagram for the main scenario of the "check-in passenger" business use case. 143
6.32 Sequence diagram with alternative scenarios of and exceptions to the "check-in passenger" business use case. 145
6.33 CPN module to represent the main scenario of the "check-in passenger" business use case. 146
6.34 CPN module to express an alternative of and an exception to the main scenario of the "check-in passenger" business use case. 148
6.35 The altenativesFF CPN module. 149
6.36 The check passport CPN module. 149

6.37 The passenger behaviour CPN module. 149
6.38 The top-most CPN module. 150
6.39 Declaration of the colour sets. 151
6.40 The top-most CPN module, with an additional use case. . . . 152
6.41 A screenshot of an animation for the check-in system. 153

Part I

Background

Chapter 1

Introduction

Summary

This chapter presents the main topics covered in this thesis: reactive software systems, validation of behavioural models, and more precisely the usage of animation to validate the requirements of a system. It also describes the problem being addressed in this thesis, identifies the aims of the thesis, enumerates the main research contributions given by this work, and presents the outline of this dissertation document.

Contents

1.1 Reactive Software Systems 4
1.2 Motivation 6
1.3 Problem Statement 7
1.4 Aims 8
1.5 Approach Taken 9
1.6 Contribution 11
1.7 Overview 12

1.1 Reactive Software Systems

Reactive software systems constitute a very wide class of systems. Reactive systems engage in stimulus-response behaviour to produce desirable effects in their environment [Wieringa 2003]. In contrast, there are the transformational systems, that exist to compute the output from an input and then terminate. In this section we introduce the main characteristics and terminology used in the class of reactive software systems that is considered in this work.

A reactive system continuously interacts with its environment, using inputs and outputs that are either continuous or discrete in time. Usually, the interaction of a reactive system with its environment is supported by a set of nonterminating processes that operate in parallel. It is expected that during the interaction the system must answer to high-priority events, even when the system is executing something else. Due to above characteristics the behaviour of reactive systems can be very complex [Manna and Pnueli 1992].

A transformational system interacts with its environment only to acquire the sufficient information to produce its output. When the system produces the output it must terminate, otherwise a failure is detected.

Real-time systems have usually reactive characteristics and the response given by a real-time system consists of a production of an output at a particular time. Thus, the correctness of a response depends on the time at which it is produced. Safety-critical systems are systems whose failure or malfunction may result in serious injury to people, or severe damage to equipment. Many of the real-time systems are also safety-critical. For example, a heart-monitoring system is a safety-critical real-time system, because the too late production of an output may cause harm to its environment. There is real-time software that is embedded in hardware systems, such as, the software that is included in telephones, or in elevator controllers. A control system is a system that enforces desirable behaviour on its environment, and in some cases it directs human actions. Examples of control systems are a manufacturing controller system, and a course managment system, that directs the actions of humans. Other examples of reactive systems are workflow management systems, enterprise resource planning systems, systems for e-commerce, and operating systems.

The examples enumerated above differ in their complexity along the dimensions of data, behaviour, and communication, and they have common characteristics that make them to be classified as reactive systems. Real-time, embedded, and control systems are reactive systems because, when

they are switched on, they enforce a certain desirable behaviour on their environment; and workflow management systems, enterprise resource planning systems systems, e-commerce systems, are reactive too because they provide desired information and enabled communication and collaboration between people or organizations in their environment. In this way we can complete the definition given above saying that "a reactive system is a system that, when switched on, is able to create desired effects in its environment by enabling, enforcing, or preventing events in the environment" [Wieringa 2003].

To design reactive systems it is crucial to consider models of their environment, where one represents the entities and the behaviour present of the environment, and the communication of the system with the environment. One can look to the system under development as a part of a network with a set of entities, with which the reactive system is interacting. The network includes devices, operators, maintenance personnel, users, software systems, and other kinds of entities. Although one can imagine that the network includes the whole world, in practice only a few entities in its immediate environment are relevant for the execution of the reactive system.

To identify the part of the world that are relevant as environment of the system it is assumed that the system and its environment exchange messages. Notice that these messages are not the messages passed between objects in an object-oriented software system. A message can be sent from the system to an entity in the environment, or it can be sent from an entity in the environment to the system. There are three aspects that characterize a message: the subject domain, the communication channel, and the function for the environment.

The subject domain defines what the message is about, that is, the entity in the environment about which the message gives some information. An entity of the environment can be a physical entity, when it has a weight and a size; a conceptual entity, when it is invisible and weightless; or a lexical item, when it is a physical entity with a meaning.

The communication channel is the path through which the message goes from the sender to the receiver, and it can introduce delays or some assumptions about the expected performance, but in this work we abstract from the physical realization of the messages.

Each message as some function for the environment. The purpose of the message can be to inform or direct the environment, or to manipulate lexical items in the system. The function of messages entering or leaving a reactive system can be used to classify a reactive system according to the function of its messages. A reactive system is said to be: informative when it answers questions, produces reports, or provides information about

the subject domain; directive when it controls, guides, or directs its subject domain; or manipulative when it creates, changes, or manipulates lexical items in the subject domain.

1.2 Motivation

Requirements engineering aims to document the user requirements, which must be an exact, unambiguous and complete description of the expectations and needs of the users. The problems resulting from a misunderstanding of the user requirements are the most expensive to correct, thus there is a need to validate the requirements early in the development process.

Validation consists on checking if a model or a system satisfies the users' expectations, assuring that their requirements are correctly captured. We differenciate validation from verification task, as the former concerns to building the right system, while the latter concerns to building the system right [Boehm 1984].

During the development of software systems, validation is a crucial activity to allow developers to be confident that they are building the correct system. One of the key issues to have a successful validation is to adopt a process where users can actively discuss the requirements of the system under development. A way to allow software designers to explain the system being developed is to permit the visualization of the design models. Two possible visualization techniques are simulation and animation. Simulation consists on showing the execution of a model, and animation is based on the visualization of a simulation in some graphical mode. The animation is usually used to convince the client that the symbolic model has some meaning in the problem domain, given a possibility to have an increasing correctness, and also completeness, of the validation task.

Models are essential for communicating and reasoning about systems and must adapt to the people and the properties of interest [Denaro and Pezzè 2004]. They are used for communicating design decisions among different stakeholders.

Behavioural models play a key role in the engineering of software-based systems. They are the basis for systematic approaches to requirements elicitation, specification, architecture design, testing, simulation, code generation, and maintenance.

A range of notations, techniques, and tools supporting behaviour modelling for these development tasks have been suggested. Underlying these notations, techniques, and tools, two complementary approaches to mod-

elling behaviour can be identified interaction-based and state-based modelling [Uchitel et al. 2005].

Interaction-based modelling focuses on the interactions (e.g. changing of messages in object-oriented systems) among the actors and the components of a system. Consequently, communication between such entities is viewed as the principal modelling construct. Interaction modelling, commonly realized using scenario and use case diagrams, provides an overall view of a system, which is particularly suited for supporting communication between project stakeholders.

To model software-based systems, the Unified Modelling Language (UML) [Fowler 2004] is the standard notation used nowadays in industry. In this work, we adopt two UML diagrams: use case diagrams and sequence diagrams. Use cases specify the set of functionalities presented by a system, and permit, due their simplicity, the dialogue between clients and developers. A sequence diagram is used to capture a behavioural scenario of a given system, which can be seen as a sequence of steps describing interactions between the actors and that system. In this work we consider that each use case is described by a non-empty set of sequence diagrams.

When developing a reactive system, which typically has an intensive behaviour and a rich set of interactions with its environment, requirements validation, before any design and implementation decisions are taken, is an important task.

The artifacts constructed in the requirements analysis are typically documents to be analysed by the stakeholders of the project. In this thesis we are tackling the problem of providing effective artifacts to allow the stakeholders to use them during the system's development, in order to facilitate the validation, in an early stage of the functional requirements being defined for the system, namely through the usage of a graphical animation.

1.3 Problem Statement

The focus of our work is in the transformation of models of behavioural scenarios into a state-based model, with the aim of facilitating the validation of the requirements of reactive systems. We assume that the requirements document includes a set of use case diagrams and the textual descriptions of the use cases. If these artefacts are not explicitly available, we consider that the document has sufficient information to obtain them. As part of our approach, the behaviour of each use case must be detailed by a collection of scenario descriptions, which can be represented by sequence diagrams.

Recently, the version 2.0 of the UML has been launched to substitute the previous versions. Sequence diagrams in UML 2.0 have many new high-level operators. Due to their simplicity, we believe that sequence diagrams of UML1.x are more adequate to capture the system's behavioural scenarios at a first stage of analysis. Thus, in the beginning we do not use all the features available in UML 2.0 (namely the high-level operators), but later in the development process, these diagrams can be aggregated into UML 2.0 sequence diagrams with all their advanced features.

Most of the works on the transformation of scenario-based models into state-based models concentrate only on the modelling of the controller part, and do not take into account the environment [Dano et al. 1997; Elkoutbi and Keller 1998], which is an important part, especially when considering reactive systems. Additionally, the obtained model usually is not parametric, i.e., it does not permit the simulation of different, but similar, scenarios obtained by just changing some initial conditions of the original scenario.

We want to explore the capabilities of the CPN modelling language in order to have a state-based and executable model of the behaviour expressed by a set of scenarios. The CPN model is constructed in such a way that it insures the following characteristics:

- **Parametric**: it allows an easy modification of the initial conditions of the scenario,
- **Environment-descriptive**: it includes the state of the relevant elements of the environment, and
- **Animation-separated**: the elements related to animation are clearly separated from the other elements in the model.

1.4 Aims

The main goal of this work is to provide a novel method into the software development process to create a graphical animation of the problem domain from a set of scenario descriptions. In particular, we aim to study for a given system how to translate a set of scenarios into a unique state-based model that represents the behaviour. The resulting model, which in this work we propose to be written in the CPN modelling language is used to coordinate an animation of the problem domain, in order to facilitate the validation activity. It is important to notice that the resulting CPN model is supposed to contain only the behaviours described in the source models,i.e., that is

there are no extra behaviours inferred from the scenarios, and consequently there are no explicit synthesized behaviours in the resulting model.

1.5 Approach Taken

The research approach taken in this work had started with the study of the state of the art, in order to clearly describe the research problem. After that, we had verified the hypothesis in the thesis finding solutions for the problem. To validate the obtained solutions we had explored three case studies already described in the literature. During the exploration of the case studies some improvements to the solution had been considered. This research approach has been presented and discussed in a doctoral symposium [Ribeiro and Fernandes 2007b]. The ideas has been also constantly discussed with our colleagues, in particular at the internal annual meeting of our department, called "Simpósio do Departamento de Informática", where each Ph.D. student has the opportunity to present the work being developed.

Our work is based on the translation of models of behavioural scenarios into a CPN model. CPNs constitute a graphical modelling language appropriate to describe the behaviour of systems with characteristics like concurrency, resource sharing, and synchronization. The CPN Tools [CPN Tools 2009; Jensen et al. 2007] is a well established tool supporting the CPN modelling language and allowing the execution of animations in accordance with the CPN model. To animate a CPN model, the BRITNeY Suite Animation Tool [Westergaard and Lassen 2006] permits the creation of an animation on top of CPN Tools, using the animation plug-in based on the SceneBeans framework [Magee et al. 2000].

Figure 1.1 sketches the general software process proposed to be followed in our approach, where rectangles represent artefacts and arrows represent activities. Please notice that the notation used in this picture is informal in its nature and is used just to give an overall view of the process. In particular, for example, the input artefacts connected to an arrow may not represent the complete information required to accomplish the corresponding activity. The same applies for the output artefacts.

The idea is that from the requirements document a set of scenario-based models can be obtained, and can be subsequently transformed into a CPN model. This CPN model, when used in conjunction with an animation specification, drives an executable animation that permits the users to perceive how the system behaves and to validate that behaviour with respect to their requirements and expectations.

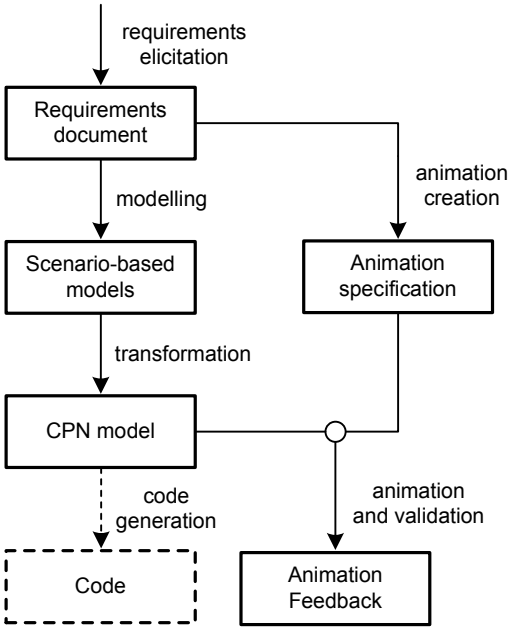


Figure 1.1: Process for the approach.

The creation of an animation is an activity that defines how each element in the problem domain is represented in the animation, namely how it is graphically depicted, the movements associated to it, the messages and the commands it sends and receives. This activity strongly depends on the requirements document, which is explicitly shown in Figure 1.1, but also, among other examples, on the creativity of the software engineer in making the animation easy to understand to the stakeholders. During this activity, it may be necessary to introduce some changes into the CPN model in order to obtain the expected animations when executing the CPN model together with the animation specification.

When inspecting the execution of the animation specification, the user usually gives valuable feedback that must be taken into account when reviewing the requirements document.

The CPN model can also be used as a formal support in the next steps of the development process. In particular, we foresee the possibility of generating implementation code from it.

Our work concentrates on two activities of the approach: transformation and animation creation (see Figure 1.1). It aims to discover how the

transformation of a set of scenario-based models into a CPN model can be automated, and which mechanisms must be used to clearly separate the CPN model and the animation specification.

In order to increase the flexibility of the animation, we aim to permit that one can change the initial conditions of the scenario being animated, to obtain a variation of the scenario. The obtained CPN model must also permit when possible the concurrent execution of various scenarios.

The approach was validated through the exploration of three examples of reactive systems. During the exploration of these systems, a conceptual tool was developed to support other applications for different examples and to be used as a formalization of the transformation.

1.6 Contribution

This section describes the contribution given by this thesis, that is the result of the execution of the approach described above (Section 1.5) in order to solve the identified problem (Section 1.3). It also presents the list of publications that has resulted from the development of this work.

The contribution of this thesis is an approach to validate, through animation, a reactive system that is described by a set of behavioural scenarios. The method transforms these scenarios into a state-based model that also includes the behaviour of elements from the environment. Tool support for the transformation is partially supported in order to illustrate that the transformation can be automated with state-of-the-art technologies.

The following papers were written based on the results obtained during this work, and they were published and presented in workshops and conferences, after being approved by a peer-review process. The papers are listed in reverse chronological order of their publication date.

1. Óscar R. Ribeiro and João M. Fernandes. *Validation of Scenario-based Business Requirements with Coloured Petri Nets.* In Fourth International Conference on Software Engineering Advances (ICSEA 2009), pages 250-255, Porto, Portugal, September 2009. IEEE Computer Society Press.
   DOI `10.1109/ICSEA.2009.45`

2. Óscar R. Ribeiro and João M. Fernandes. *On the Use of Coloured Petri Nets for Visual Animation.* In Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2007), pages 223–241, Århus, Denmark, October 2007.

URL http://www.daimi.au.dk/CPnets/workshop07/cpn/papers/
(This paper has been cited by 1 work from other authors)

3. Óscar R. Ribeiro and João M. Fernandes. *Validation of Reactive Software from Scenario-based Models.* In Second Software Engineering Doctoral Consortium (SEDES 2007) at QUATIC 2007 pages 213–217, Lisbon, Portugal, September 2007. IEEE Computer Society Press.
DOI 10.1109/QUATIC.2007.33

4. Óscar R. Ribeiro and João M. Fernandes. *Translating Synchronous Petri Nets into PROMELA for Verification of Behavioural Properties.* In Second IEEE International Symposium on Industrial Embedded Systems (SIES 2007), pages 266–273, Lisbon, Portugal, July 2007. IEEE Computer Society Press.
DOI 10.1109/SIES.2007.4297344
(This paper has been cited by 3 works from other authors)

5. João M. Fernandes, Simon Tjell, Jens B. Jørgensen, and Óscar R. Ribeiro. *Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net.* In Sixth International Workshop on Scenarios and State Machines (SCESM 2007) at ICSE 2007, Washington, DC, USA, May 2007. IEEE Computer Society Press.
DOI 10.1109/SCESM.2007.1
(This paper has been cited by 6 works from other authors)

6. Óscar R. Ribeiro and João M. Fernandes. *Some Rules to Transform Sequence Diagrams into Coloured Petri Nets.* In Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006), pages 237–256, Århus, Denmark, October 2006.
URL http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/
(This paper has been cited by 5 works from other authors)

1.7 Overview

This section gives an overview about how this dissertation document is organized. The structure of the document is described, and for each chapter a brief description is given.

This dissertation document is divided into two distinct parts:

Part I includes chapters 1 to 4 and presents the background of this thesis;

Part II includes the chapters 4 to 8 and presents the contribution of this thesis.

The chapters in this thesis are organized as follows.

Chapter 2 introduces three distinct types of models to represent the behaviour of a reactive software system. Firstly, we present the use case model, part of the UML notation, that is used to capture the actions produced by the system as they appear to its users. Secondly, the interaction models of the UML, in particular the sequence diagrams, are described in detail namely by discussing their metamodel in the UML 2.0. Thirdly, the CPN modelling language is presented. CPN is a formal modelling language, suitable for describing the behaviour of systems with characteristics like concurrency, resource sharing, and synchronization. The tool support for this modelling language is also introduced.

Chapter 3 introduces the requirements engineering area. The fundamental process followed in a requirements engineering project is detailed, including the description of models and actors involved in the process, and how it fits in the overall software engineering process. Afterwards, we briefly describe some approaches for requirements elicitation, which is concerned in defining where software requirements come from and how the software engineer can collect them. The chapter ends with a description of approaches for requirements validation, including some graphical animation techniques.

Chapter 4 introduces a set of rules to transform a set of sequence diagrams of UML 2.0 into a CPN model. It is detailed how to transform plain sequence diagrams, which define the messages in the sequence diagrams and the order between them, in terms of constructs of the CPN modelling language. Afterwards, one defines how some of the high-level operators of the sequence diagrams are transformed into elements of the CPN model.Additionally, are shown the results of applying these ideas to some illustrative examples. The tool support available for the rules to transform sequence diagrams into a CPN model is described.

Chapter 5 describes some guidelines that we suggest to be taken into account when enriching a CPN model for animation purposes. These guidelines aim to ensure that the obtained CPN model: (1) allows the easy modification of the initial conditions of scenarios; (2) includes

a description of the state of the relevant entities in the environment; and (3) implies that the elements related to the animation are clearly separated from the other ones.

Chapter 6 presents the case studies that were considered in this thesis: (a) the reactor system; (b) the elevator controller system; and (c) the check-in system in an international airport. These are different examples of reactive systems. The reactor system does not interact with human actors. In the elevator system there are human actors that interact with the system through the interface elements. Human actors are an important part of check-in system, thus it is crucial to consider their behaviour. The rules and the guidelines introduced in previous chapters are discussed and are exemplified in the context to the three case studies.

Chapter 7 presents the state-of-the-art of the research fields considered in this work. The chapter starts with the analysis of some efforts on turning the use case descriptions into a more formal representations. After that, we consider some methods to directly generate a state-based model from a scenario-based descriptions, where there are no explicit reference to states of the system. The use of Petri nets as a medium to a synthesise scenarios describing a system behaviour is analysed. This chapter ends with the description of some efforts that have been presented to improve the way that formal specifications are presented to the users, in particular when these specifications represent software requirements.

Chapter 8 presents a summary of the contributions resulting from the work described in this dissertation document, and points out several ideas for future work.

Chapter 2

Behavioural Models

Summary

This chapter introduces three distinct types of models to represent the behaviour of a reactive software system. Firstly, we present the use case model, part of the UML notation, that is used to capture the actions produced by the system as they appear to its users. Secondly, the interaction models of the UML, in particular the sequence diagrams, are described in detail namely by discussing their metamodel in the UML 2.0. Thirdly, the CPN modelling language is presented. CPN is a formal modelling language, suitable for describing the behaviour of systems with characteristics like concurrency, resource sharing, and synchronization. The tool support for this modelling language is also introduced.

Contents

2.1 Use cases **16**
2.2 Interactions **17**
2.3 Coloured Petri Nets **25**

2.1 Use cases

Use cases capture a set of actions produced by a system as they appear to users outside the system. Each use case is intended to provide a unit of coherent behaviour without revealing the internal structure of the system. The behaviour present in a use case is specified by defining the interactions of the actors with the system. An *actor* is an idealized user of the system, which includes not only the humans, but also computer systems and processes. The overall functionality of a system is partitioned over a set of use cases, each one representing a meaningful piece of functionality for the involved actors. An interaction with actors is described as a sequence of messages between the system and one or more actors, and it includes the normal behaviour, as well as some possible variants of the normal sequence, such as alternate sequences, exceptional behaviour, and error handling. The goal is to put a piece of coherent functionality in all use case's variations (including the error conditions).

The behaviour of a use case is sometimes described by a piece of informal text. It can also be specified in other ways: by an activity specification, by an attached state machine, by an interaction describing legal sequences, or by pre- and post-conditions. An execution of a use case is called a *use case instance*. A use case describes potential behaviour, thus "*behaviour can be illustrated, but not formally specified, by a set of scenarios*" [Object Management Group 2007]. Usually, at an early stage of the development process, this is sufficient.

In this way, use cases capture who (actor) does what (interaction) with the system, without detailing the system's internals. A complete set of use cases specifies all the different ways to use the system, and therefore defines the behaviour of the system, bounding its scope.

The *subject* of a use case is the system under consideration, that may include a physical part and any other elements, such as a component, a subsystem, or a class that may have an impact on the system's behaviour. Each use case specifies a specific way for users to interact with the system, and it constitutes a unit of useful functionality that is provided to its users. This functionality is usually initiated by an actor, and when it is completed the use case is also completed.

A use case diagram shows the relationships among actors and use cases within a system. A use case diagram includes a set of actors, a set of use cases enclosed by a subject boundary (a rectangle), associations between the actors and the use cases, relationships among the use cases, and generalizations among the actors. In Figure 2.1 we have an example of a sequence diagram

of the "telephone catalog" system. The diagram is decorated with textual labels identifying the elements of the UML 2.0 metamodel The textual labels are in italic, and their connection with use case diagram's elements is done by an arrow.

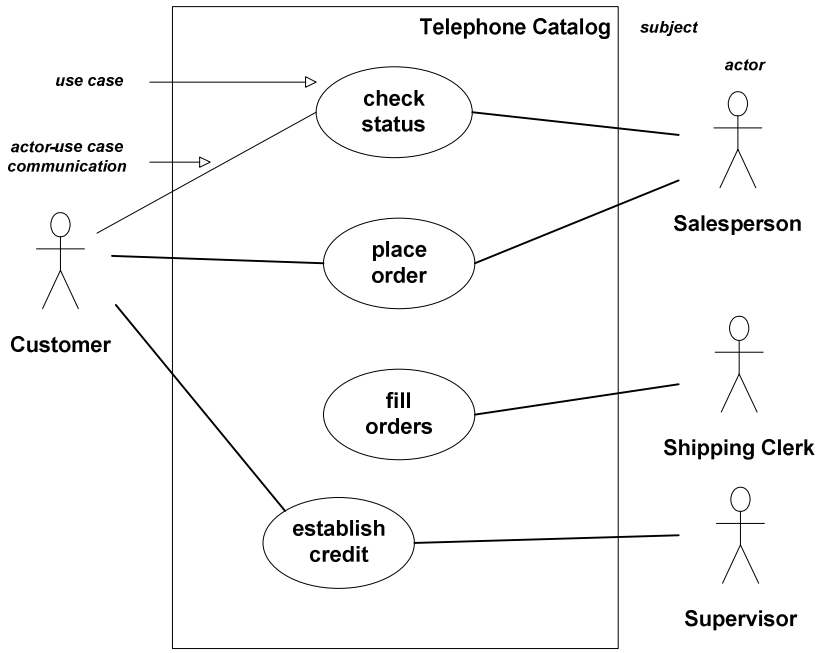


Figure 2.1: An example of a use case diagram.

A use case is shown as an ellipse with its name inside (or below) the ellipse. The system boundary is represented by a rectangle containing the use cases. The actors are placed outside the rectangle to show that they are external. The connection of actors with the use cases they participate is represented by a line.

2.2 Interactions

2.2.1 Introduction to interactions

Interactions are a mechanism for describing systems that can be understood and produced, at varying levels of detail, by both professionals of computer systems design, as well as potential end users and stakeholders of the system under development. They can be used at different stages of the development

process and they are used to improve the understanding of an interaction situation. Interactions are also used during the more detailed design phase, where the precise inter-process communication must be described according to formal protocols. When testing is performed, the traces of the system can be described as interactions and compared with those iterations of the earlier phases.

The behaviour in an interaction is represented by the exchanging of messages among the participating objects, and by the ordered sequence of the events (such as the sending and the receipt of a message). Interactions are centred in a set of cooperating objects; this focus is complementary to the description of behaviour in state machines that is centred on individual objects.

A message represents a flow of control, possibly with information, from a sender to a receiver, that is, it constitutes a one-way communication between two objects. Values can be carried from the sender to the receiver through arguments in messages. There are two kinds of messages:

- a signal - an explicit, named, asynchronous interobject communication; or
- a call - the synchronous or asynchronous invocation of an operation with a mechanism for later returning control to the sender of a synchronous call.

An event is something that happens during the execution of a system, and that is important to model. It has a location in time and space, and it occurs instantaneously (no duration). Both messages and events are used in various UML behavioural diagrams, such as activity diagrams, sequence diagrams and state machine diagrams.

Diagrams provide a visual notation for the UML modelling concepts. There exist two kinds of diagrams to express the sequential order of the messages and the events:

- the sequence diagram, that focuses on the time sequences of the messages;
- the communication diagram, that focuses on the relationships among the objects that exchange the messages.

2.2.2 Sequence diagrams

This section describes how an interaction can be represented by a sequence diagram. Figure 2.2 presents an example of a sequence diagram, which

is decorated with textual labels identifying the elements of the UML 2.0 metamodel. The textual labels are in italic, and their connections to the sequence diagram's elements are done by an arrow.

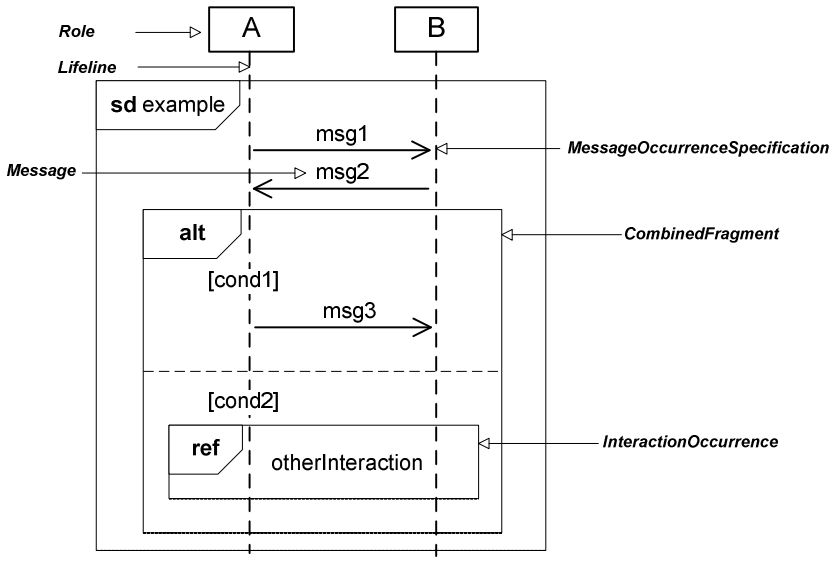


Figure 2.2: An example of a sequence diagram.

An interaction is displayed in a sequence diagram as a two-dimensional chart. The vertical dimension is the time axis. The horizontal dimension shows the roles that represent individual objects and actors in the collaboration. Each role is represented by a vertical column containing a head symbol and a vertical line, called a lifeline, which is displayed as a dashed line.

In Figure 2.2 there are two lifelines corresponding to roles `A` and `B`. Between the lifelines of an interaction there are messages that come from a sender to a receiver. A message is shown as a line with an open arrowhead which indicates its direction. In the figure we have three messages, that are called `msg1`, `msg2` and `msg3`. Message `msg1` goes from `A` to `B`, and it occurs before the message `msg2` that goes in the opposite direction (i.e., from `B` to `A`).

As the name indicates, a sequence diagram shows the flow of control as a sequence of messages. To show more complex flows of control it is necessary to use high-level interaction fragments, called combined fragments. Each

combined fragment has a keyword associated with it, and some subfragments, which are called interaction operands. The keyword in a combined fragment defines the interpretation of the control flow in each operand, and the number of operands to be considered. There is also a mechanism, called interaction use, to introduce in a sequence diagram a reference to another interaction. Roughly, we can say that this mechanism is similar to the concept of subroutine in programming languages. In Figure 2.2 we can find a combined fragment with the keyword `alt`, that represents an alternative between the two operands in the example. The second operand of the `alt` combined fragment uses the interaction use mechanism to refer to another interaction called `otherInteraction`.

2.2.3 Metamodel for UML2 sequence diagrams

In this subsection we present the metamodel of the interactions, described in the "UML Superstructure Specification" (version 2.1.2) [Object Management Group 2007]. The sending and the reception of a message, called message occurrence specifications, represent events occurring in the interaction and constitute a particular kind of event occurrences. A sequence of event occurrences is a called a trace. A semantic of an interaction is given by defining the set of valid traces, and possibly also by identifying the set of invalid traces.

In this thesis we are considering that the messages in the sequence diagrams are synchronous, thus there exists a unique event associated with the execution of a message.

To show more complex flows of control, it is necessary to use high-level interaction fragments, called combined fragments. Each combined fragment has a keyword associated with it, and some subfragments, which are called interaction operands. The keyword in a combined fragment defines the interpretation of the control flow in each operand, and also the number of operands to be considered.

A combined fragment is shown as an outline rectangle, and inside a pentagon in the upper left corner of the rectangle there is the corresponding keyword.

The metamodel for the interactions is depicted in [Object Management Group 2007, Section 14.2], where the abstract syntax for interactions is divided along six diagrams, each one showing different aspects of an interaction.

The `Interaction` element is the main element when considering the modelling of an interaction (see Figure 2.3). Each `Interaction` is associated

with: a set of `InteractionFragments`, a set of `Lifelines`, and a set of `Messages`.

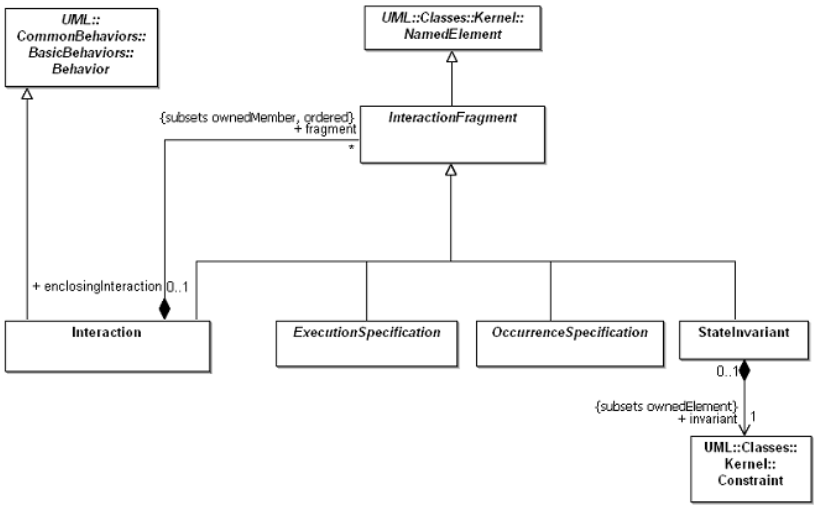


Figure 2.3: Part of UML2 metamodel for Interactions [Object Management Group 2007, p. 460].

An `Interaction` is a subclass of `InteractionFragment`. In this way, it is possible to have many `Interactions` associated with an `Interaction`, and these `Interactions` can have different kinds as detailed below. An `interaction fragment` is a piece of an `interaction`, because it is an abstract notion of the most general interaction unit.

A `Lifeline` represents an individual participant of an `Interaction`, and it is covered by an ordered set of `InteractionFragments` (see Figure 2.4). The connection of `Lifeline` with a set of `InteractionFragments` is redefined by two other connections: one with an ordered set of `OccurrenceSpecification` and another with a set of `StateInvariants`, where both are specializations of `InteractionFragment`.

A `Message` flows among a kind of `OccurrenceSpecification`, called `MessageOccurrenceSpecification` (see Figure 2.5).

The basic semantic units in interactions are the `OccurrenceSpecifications`, which are ordered along a lifeline. The meaning of interactions is given by the sequences of occurrences specified by the `OccurrenceSpecification` (see Figure 2.6). `StateInvariants` are runtime constraints on the participants of the interaction; this work does not consider the possibility of having `StateInvariants` in the lifelines.

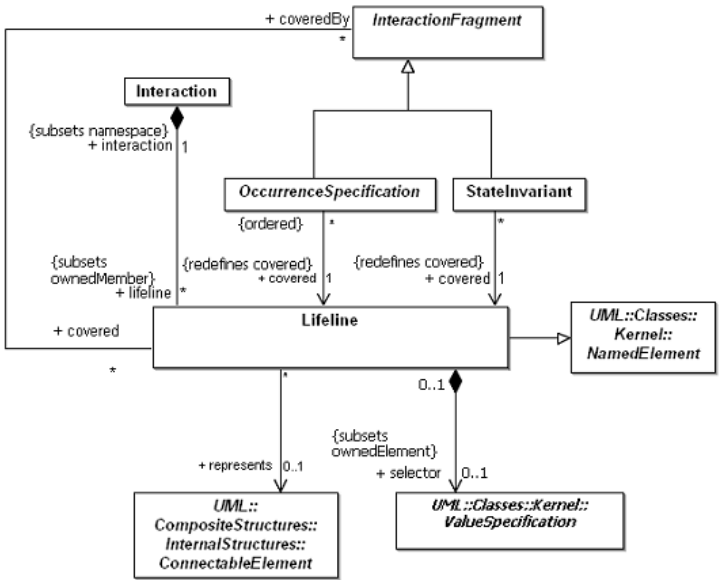


Figure 2.4: Part of UML2 metamodel for Lifelines [Object Management Group 2007, p. 461].

One of the kinds of `Interaction` is the `CombinedFragment`, which consists of an `InteractionOperator` and a set of `InteractionOperands` (see Figure 2.7).

Each `InteractionOperand` can be a plain `Interaction` or again a `CombinedFragment`. In this way, a `CombinedFragment` defines an expression containing `InteractionFragments`. `CombinedFragments` allow the user to describe a number of traces in a compact and concise way.

The number of operands embedded in a `CombinedFragment` and its semantic depend on the `InteractionOperator`, which is defined by one of the following keywords:

sd is used to indicate the principal frame of the sequence diagram. Usually, it is followed by the name of the diagram;

ref is used to indicate a reference to another fragment of interaction;

opt specifies one operand with a guard condition associated with it. If the guard evaluates to true the subfragment is executed; otherwise it is not executed;

break specifies one operand with a guard condition associated with it. The

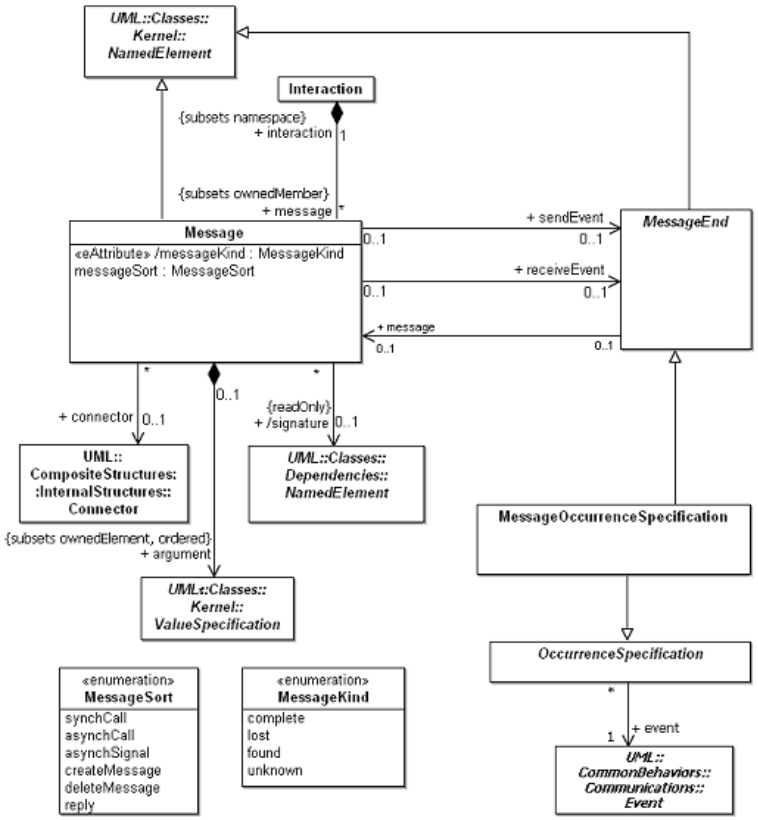


Figure 2.5: Part of UML2 metamodel for Messages [Object Management Group 2007, p. 462].

behaviour of the operand is executed if the guard is true, and in this case the remainder of the fragment is not executed. Otherwise, the execution continues normally;

alt specifies that the fragment represents a choice between multiple operands. There is a guard associated with each operand (the absence of the guard is interpreted as true), whose evaluation defines which operand is executed. If more than one of the guards are evaluated to true, the choice may be non-deterministic;

par indicates that the fragment represents a parallel merge between the behaviours of the operands. The events from the parallel subfragments can interleave in any possible order;

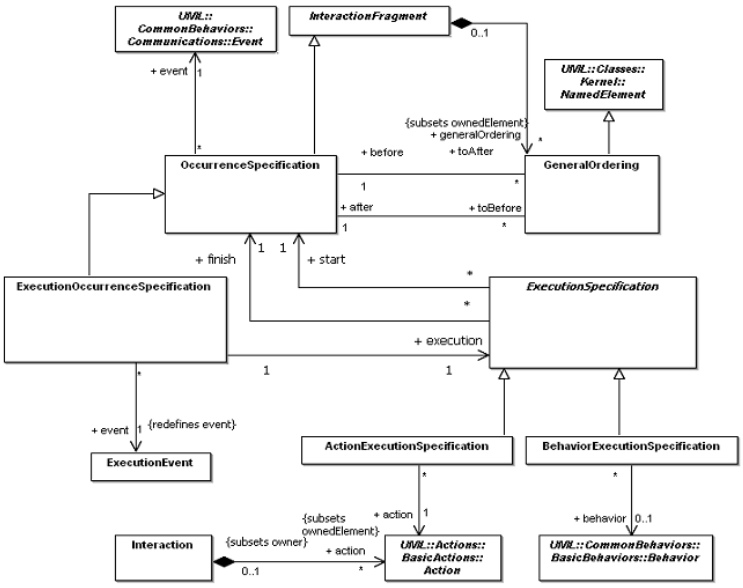


Figure 2.6: Part of UML2 metamodel for Occurrence Specifications [Object Management Group 2007, p. 463].

loop indicates an interaction fragment that shall be repeated a given number of times. It has a guard associated with it, and also a minimum and a maximum counters. The interaction fragment is executed (at least the minimum count and no more than the maximum count) until the guard evaluates to false;

seq indicates the weak sequencing of the operands in the fragment, which is selected by default. The weak sequencing maintains the order inside each operand, and the events on different operands and different lifelines may occur in any order;

strict specifies that messages in the fragment are fully ordered.

There are some other keywords that are not covered in this work such as *neg*, *assert*, *consider*, and *ignore*.

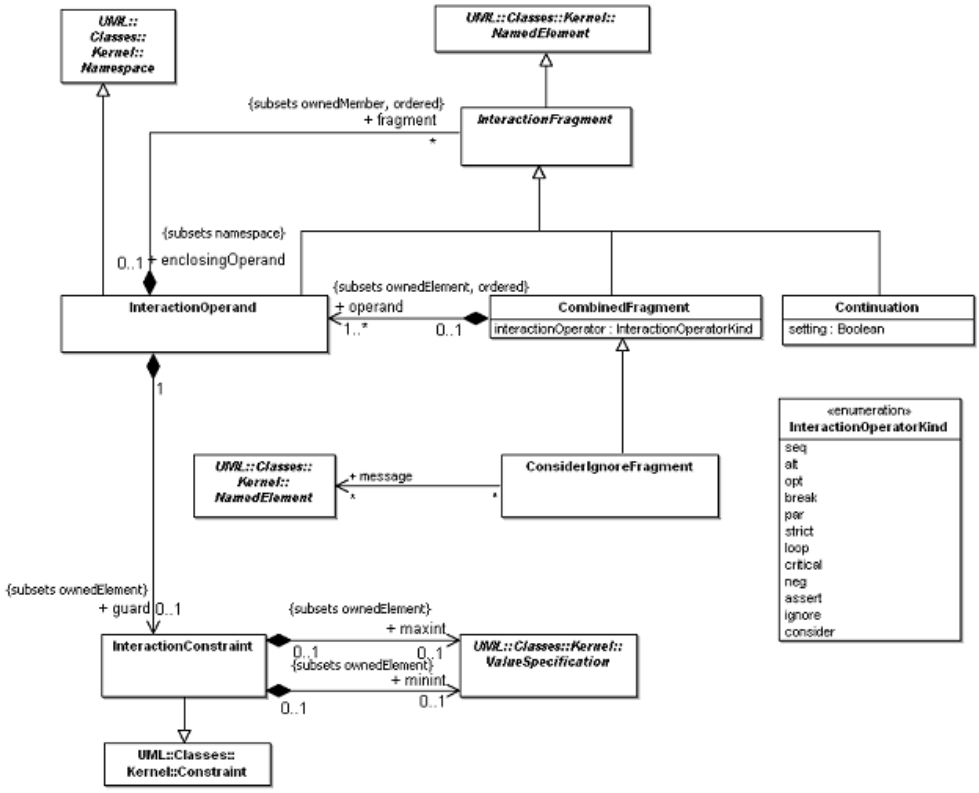


Figure 2.7: Part of UML2 metamodel for Combined Fragments [Object Management Group 2007, p. 464].

2.3 Coloured Petri Nets

This section presents the CPN modelling language. The section starts by an informal introduction of CPN modelling language. After that it is shown how to create a CPN model to represent a pratical problem. Then, a formal definition of the CPN modelling language is presented. Finally, the main features of CPN tools is introduced.

Petri Nets (PNs) [Reisig 1985] constitute a suitable model of computation for expressing concurrent systems, due to their extensive body of results, both theoretical and practical. PNs have shown to be a powerful technique to specify and model the behaviour of systems, where concurrency, resource sharing, and synchronisation are important issues to take into account.

CPN is a variant of Petri nets, suitable for describing the behaviour

of complex systems. The CPN language is supported by CPN Tools that facilitate construction, editing, execution, and analysis of the CPN models.

CPN models are typically used to represent problems of distributed algorithms [Reisig 1998], embedded systems [Adamski et al. 2005; Yakovlev et al. 2000], communication protocols [Billington et al. 2004] and data networks [Billington et al. 1999]. There are some other applications of CPNs to more generic systems, such as business process and workflow modelling [van der Aalst and van Hee 2004], and manufacturing systems [Desrochers and Al-Jaar 1994]. Industrial applications of CPNs are detailed in [Jensen and Kristensen 2009, chap. 14].

2.3.1 Basic concepts

The CPN language provides an explicit description of both states and actions, and gives a modelling convenience corresponding to a high-level programming language with support for data types, modules, and hierarchical decomposition.

A CPN model is a graphical structure, composed of *places*, *transitions*, *arcs*, *tokens*, and *inscriptions*, supplemented with declarations of *data types*, *variables*, and *functions*. An example of a CPN model is shown in Figure 2.8.

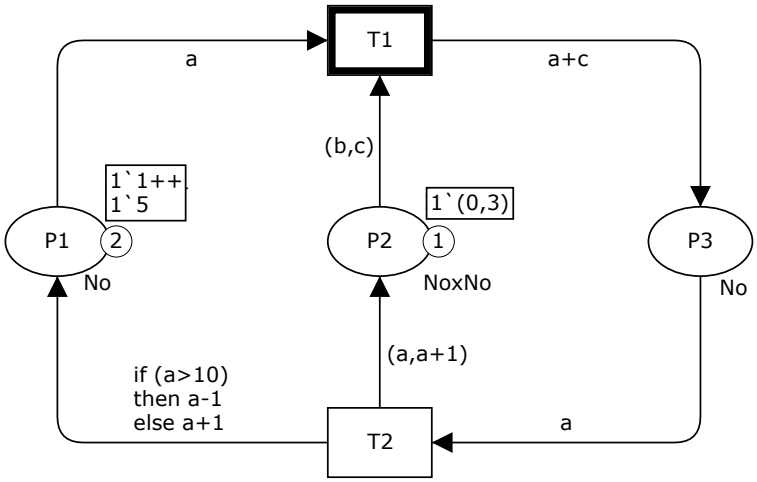


Figure 2.8: A small CPN model.

The tokens may have complex data values. In CPNs the data values are called as *colours*, and data types are called *colour sets*. The use of functions and expressions to handle data values permits the complexity of a model to

be divided among graphics, inscriptions, and declarations.

The inscriptions and declarations are defined using the functional programming language CPN-ML which is based on Standard ML [Standard ML 2006; Ullman 1998]. The CPN model in Figure 2.8 uses the CPN-ML code in Figure 2.9.

```
1 colset UNIT   = unit;
2 colset INT    = int;
3 colset BOOL   = bool;
4 colset STRING = string;
5 colset No     = int;
6 colset NoxNo  = product No*No;
7 var a, b, c : No;
```

Figure 2.9: CPN-ML code used in the CPN model presented in Figure 2.8.

In CPN tools, colour sets are defined using the CPN-ML keyword colset. In the example the colour set No is defined to be equal to the integer type int (line 5). The colour set NoxNo is defined to be the product of the type No with itself (line 6). There are also variables containing elements of the defined colour sets, for example variables a, b and c are defined to have the colour set No (line 7).

Places are drawn as ellipses and hold multisets (bags) of *tokens*. A place models a local state, given by its tokens. The global state of a model is the union of all local states. Each place has an associated colour set, indicated by an inscription near the place, that specifies the kind of tokens that it may contain. For example, the places P1 and P3 in Figure 2.8 can both contain tokens of the type No. The place P1 has two tokens, indicated by the circled "2", and the place P3 has no tokens.

A multiset (or bag) is a generalization of a set, where each element can appear more than once. A place of a CPN model contains a multiset whose elements have the token colour defined for the place. The construction of a multi-set are supported by two operators ++ and `. The operator ++ takes two multisets as arguments and returns their union (sum). The operator ` takes an integer (non-negative) as left argument specifying the number of occurrences of the element represented by the right argument. The absence of an inscription specifying the initial marking means that the place initially contains no tokens. This is the case for place P3.

There is an inscription written above the place that sets the *initial marking* of the place. For example, the inscription 1`1 + +1`5 at the upper right

side of place P1 specifies that the initial marking of this place consists of two tokens representing two integer numbers, where one token has the colour (value) 1, and the other has the colour 5.

This means that in the CPN language the pattern of behaviour is expressed only once by the structure of the CPN module, and multiple instances operating concurrently are modelled by tokens with individual identification.

Transitions model behaviour and are drawn as boxes. A transition is connected to *input places* and *output places* by *arcs*. Each transition has a name that has no formal meaning, but is very important for the readability of the model. When a transition fires, it removes tokens from its input places (those places that take an arc going to the transition) and it adds tokens to its output places (those places that have an arc coming from the transition). The colours of the tokens that are removed from input places and added to output places when a transition occurs are determined the textual inscriptions positioned next to the individual arcs.

The *guards* are associated to transitions and, by default, are located near the top left corner of the rectangle representing the corresponding transition.

In the CPN model of Figure 2.8 there are two transitions, T1 and T2. The T1 transition has places P1 and P2 as its input places and place P3 as its output place. When T1 fires, one token is removed from each of its input places and a new token is added to its output place. A transition is said to be *enabled* (i.e., ready to fire), when it is possible to consume a collection of tokens from its input places that complies with the restrictions expressed by the inscriptions on the arcs connecting these places to the transition.

The T1 transition is enabled when:

- there is at least one token in the P1 place, and
- there is at least one token in the P2 place.

Thus, the enabling of transition T1 does not depend on the values of the tokens in its input places, and the value of the token to be added to the output place P3 is the sum of variables a (token in place P1) and c (the second component of token in place P2).

A CPN model can be organised as a set of hierarchically related *modules*, in a similar way as programs are constructed from modules. *Substitution transitions* (of which there are none in Figure 2.8) constitute the basic mechanism for hierarchically structuring a CPN model. A substitution transition, graphically represented by a doubled-edged box (see, e.g., Figure 6.38) is a transition that stands for a whole module of the CPN

structure. A substitution transition in a super-module is connected to its sub-module via places in the two modules, which are conceptually linked together. Substitution transitions serve the same role in CPN models with respect to the structuring of models as superstates do in statecharts.

The CPN modelling language has a mathematical definition of its syntax and semantics. In this way, one can say that the CPN models are formal. This means that they can be used to verify system properties, i.e., to prove that certain desired properties are fulfilled or that certain undesired properties are guaranteed to be absent. Verification of system properties is supported by a set of state space methods [Jensen and Kristensen 2009, Chapter 8].

The practical application of CPN modelling and analysis relies heavily on the existence of computer tools supporting the creation and manipulation of models.

2.3.2 Creating a CPN model

In this subsection, we illustrate how to create a CPN model that captures the behaviour given by an example described below. Different CPN models are created for the same example to show different approaches of modelling. The system to be considered here comes from the analysis of the actions that can be taken by a person who is at the hall of an elevator car at a given floor of the building and wants to travel to another floor. To simplify the example used in this subsection only a part of this system is considered. Firstly, we consider only two floors, and at each floor there is only one person who can find buttons to select the possible travel direction (in the first floor it is obviously only possible to travel in the up direction, and in the second floor we assume that it is possible to travel in the up direction). An overview of this example is depicted in Figure 2.10. In the first floor (floor 1) there is a person, called Bob, who can only press the button to go up, and in the second floor (floor 2) there is also a person, called Jim, who can press either the button to travel down or the button to travel up.

Due to the simplicity of this system, it is easy to detail its behaviour, enumerating all the six accessible states:

1. None of the persons have pressed a button;
2. Bob has pressed the button (to travel up), and:
   (a) Jim has not pressed any button;
   (b) Jim has pressed the button to travel up;

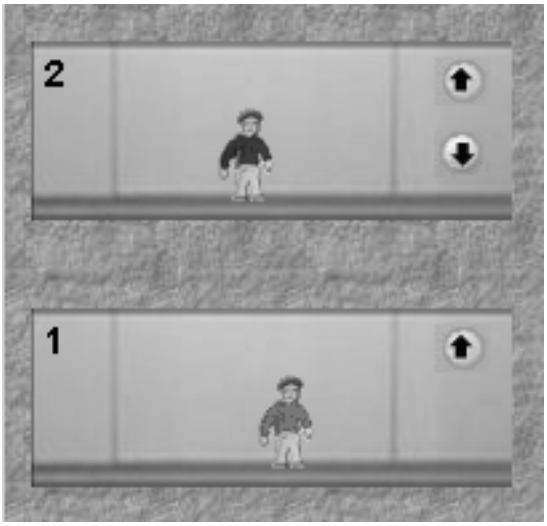


Figure 2.10: Sketch of the options that passengers at each floor have to call an elevator car.

(c) Jim has pressed the button to travel down;

3. Jim has pressed the button to travel up, and Bob has not pressed any button;

4. Jim has pressed the button to travel down, and Bob has not pressed any button;

In a CPN model the union of all places represent the global state of the model. Inside each transition and place we can find a label that may be used to describe its role in the model. Figure 2.11 presents a CPN model for the system depicted in Figure 2.10, where we can identify two independent parts, each one related to one of the passengers. On the left hand side we have the model related with actions of Bob, who has only the button to go up to press. The transition `Bob presses Up at 1st Floor` represents the action of Bob, which has as input place `Bob is at 1st Floor`, whose token represents the fact that Bob is at the first floor, and this place initially has one token in it. The colour being used in this CPN model uses only the colour set `UNIT`, that represents the data type with only one element, represented by the symbol `()`.

On the right hand side there is the part of the model related with actions of Jim, who has the possibility to press one of the two buttons in the hall.

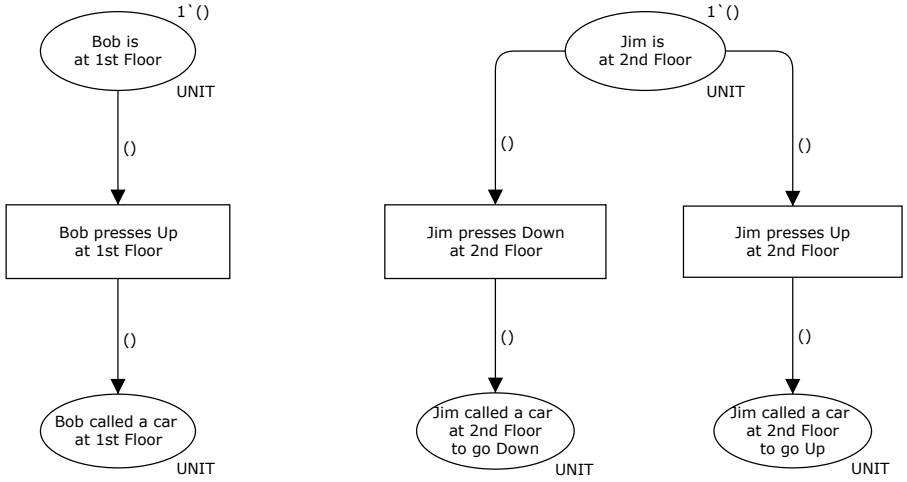


Figure 2.11: CPN model that for the example, that uses only the colour set UNIT.

The transition `Jim presses Down at 2nd Floor` represents the pressing of the button to travel down by Jim, and the transition `Jim presses Up at 2nd Floor` represents the pressing of the button to travel up by Jim. The place `Jim is at 2nd Floor` represents the fact that Jim is at second floor and it is an input place for both transitions. The existence of distinct output places to those two transitions to capture the fact that Jim is supposed to only press one of the two buttons. In this CPN model, the complexity of the model is reflected in the number of nodes, which implies that the colour sets being used are the most simple ones. One can see that the `UNIT` colour set gives to tokens the same expressive power as the one used in other models of Petri nets, such as place/transitions Petri nets [Reisig 1985]. Introducing more complexity on the colour sets, and subsequently increasing the complexity of the arc's inscriptions, results on a CPN model for the same example with a lower number of transitions and places, due to the more abstract role in the CPN model of some transitions and places. Figure 2.12 presents a CPN model with the same purposes as the one in Figure 2.11, using some colour sets that are more complex than `UNIT` colour set.

The main change introduced in this version is the usage of the same transition to be executed for different floors. In this case we are considering that each token in the place `Passengers at Floor` contains the data about

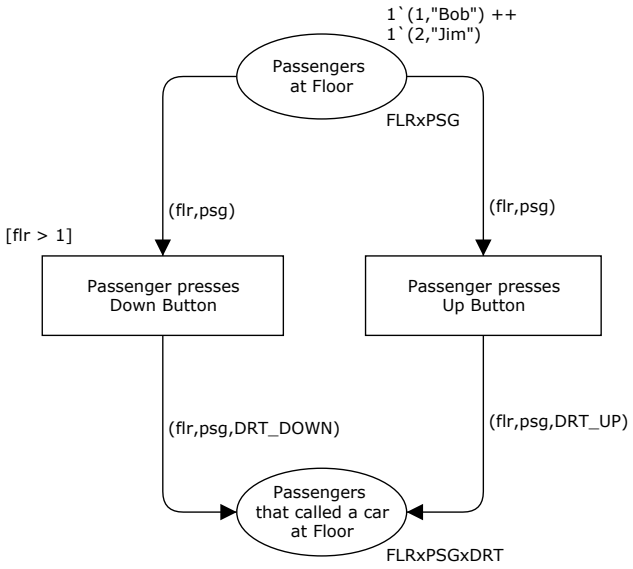


Figure 2.12: Example of CPN model for the example.

one floor and data about the passenger in that floor. The transitions are used to represent the actions of a passenger, in a given floor, to press the button to go up, or the button to go down. The first floor has only a button to call a car to go up; this property is introduced in the model using the guard $[flr > 1]$, that is the transition `Passenger presses Down Button` is not enabled for the first floor. Each token in the place `Passengers that called a car at Floor` has the data about the considered floor, the respective passenger, and the direction selected by the passenger.

To declare colour sets to be used in the places and variables, in the transition's guards and in the arc's inscription, it is necessary to consider some additional code in the CPN-ML programming language. The CPN model in Figure 2.12 considers the CPN-ML code in Figure 2.13. In line 1, the `FLOOR_NUMBER` colour set (using the keyword `colset`) is defined as an integer. In line 2, the `PASSENGER` colour set is defined as a string. In lines 3 and 4, the `DIRECTION` colour set is defined as a colour set that has two elements: the `DRT_UP` to identify the up direction, and the `DRT_DOWN` to identify the down direction. In lines 5 and 6, the `FLRxPSG` is defined as the product of `FLOOR_NUMBER` and `PASSENGER` colour sets, whose elements are pairs of a number of a floor, and the string identifying the passenger in that floor. In lines 7-9, the `FLRxPSGxDRT` is defined as the product of

```
1  colset FLOOR_NUMBER = INT;
2  colset PASSENGER = STRING;
3  colset DIRECTION = with DRT_UP
4                        | DRT_DOWN;
5  colset FLRxPSG = product FLOOR_NUMBER
6                         * PASSENGER;
7  colset FLRxPSGxDRT = product FLOOR_NUMBER
8                              * PASSENGER
9                              * DIRECTION;
10 var flr : FLOOR_NUMBER;
11 var psg : PASSENGER;
```

Figure 2.13: CPN-ML code used in the CPN model in Figure 2.12.

`FLOOR_NUMBER`, `PASSENGER` and `DIRECTION` colour set. In lines 10 and 11, there are the declaration, using the keyword `var`, of variables `flr` as a `FLOOR_NUMBER` colour set, and `psg` as a `PASSENGER` colour set.

The initial marking is introduced in the place `Passengers at Floor` using the following multi-set $1\text{`}(1,'\texttt{Bob}') + +1\text{`}(2,'\texttt{Jim}')$ that indicates that Bob is at floor 1 and Jim is at floor 2.

When considering this initial marking the transition `Passenger presses Down Button` occurs with the token of the data related with the Jim, and the transition `Passenger presses Up Button` occurs either with the token for Jim, or the token for Bob.

This model can easily scale up with respect to the considered number of floors (each one is identified by an unique integer) introducing more tokens in the place `Passengers at Floor` to represent passengers that can possibly be present in the existing floors. When adding a new floor it is necessary to express which buttons are present in the hall, and this may be achieved in the CPN model by using the guards in the transitions, which are used to restrict the buttons that are present at each floor. When interpreting this CPN, it is not explicit which buttons are present in each floor. One only becomes aware of that by inspecting the expressions in the guards. Next we present a CPN model with the same purpose using less graphical elements to illustrate the capability of the CPN-ML expressions. Figure 2.14 presents a CPN model that is intended to capture the same behaviour as the CPN models presented above. The main change introduced here is the consideration of only one transition called `Passenger presses a Button` to capture the action of pressing a button by a passenger, that includes

either pressing the button to go up or pressing the button to go down.

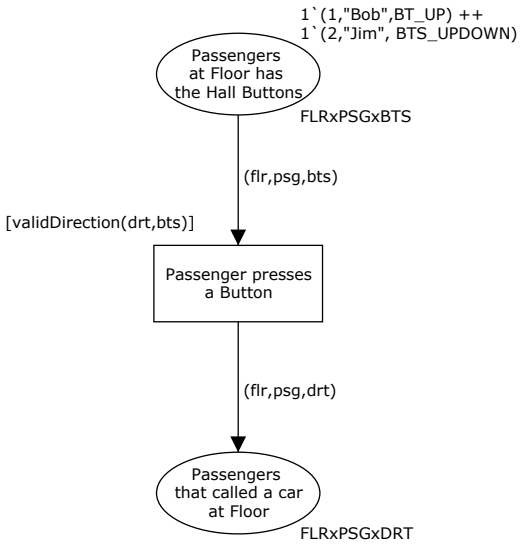


Figure 2.14: Example of CPN model.

Figure 2.15 presents the CPN-ML code used in the CPN model in Figure 2.14. In line 1, the `HALLBUTTON` colour set is defined using the keyword

```
1 colset HALLBUTTON = with BTS_UPDOWN | BT_UP | BT_DOWN;
2 colset FLRxPSGxBTS = product FLOOR_NUMBER *
3                              PASSENGER    *
4                              HALLBUTTON ;
5 var bts : HALLBUTTON;
6 var drt : DIRECTION;
7 fun validDirection(UP,BTDOWN)= false
8   | validDirection(DOWN,BTUP)= false
9   | validDirection(_,_)      = true;
```

Figure 2.15: CPN-ML code used in the CPN model in Figure 2.14.

`with` meaning that a value of this colour set can have one of the three indicated values. This colour set represents the three possibilities for the configuration of the buttons in the hall. In lines 2-4, the `FLRxPSGxBTS` is defined as the product of the `FLOOR_NUMBER`, the `PASSENGER`, and the `HALLBUTTON` colour sets. This colour set is used in the places of the CPN model, allowing one to identify the passenger in it, and the indication of the available

buttons at the floor. The predicate `validDirection` (lines 7-9) is used to evaluate if a direction to travel is valid in a given floor.

2.3.3 Tool support

CPN Tools constitutes the computer tool support for the pratical application of the CPN modelling language, allowing the creation and manipulation of models [CPN Tools 2009; Jensen et al. 2007]. In particular, the tool allows the editing, the simulation, the state space analysis, and the performance analysis of CPN models. It includes mechanisms to directly work on the graphical representation of the elements present in the CPN model.

CPN Tools includes a visualisation package called *The BRITNeY Suite Animation Tool* that facilitates the usage of application domain graphics on top of CPN models [Westergaard and Lassen 2006]. The tool is implemented in Java allowing to use the animation plug-in based on the SceneBeans framework [Pryce and Magee 2007].

The functionalities of CPN Tools are divided in two main components: an editor and a simulator. BRITNeY is integrated with CPN Tools, through a communication with the simulator using a standard remote procedure call protocol. The visualization in BRITNeY is updated when a transition occurs in the simulation of the CPN model. The correspondence between the transition in the CPN model and the change to introduce in the visualization, is done by calling a command available in the visualization in special transition inscriptions called *code segments*. A code segment is executed when the transitions that it belongs to occurs. Code segments are allowed by CPN Tools, and they consist of input, output, and action parts. The input and output parts make it possible to receive input from the model and to send feedback to the model, respectively. This allows a command to be invoked with values parametrized by tokens and to generate new tokens from the result of executing a command on the visualization.

BRITNeY has already been used in different contexts, such as to animate a network protocol [Kristensen et al. 2005], or to animate a workflow processes with the purpose of requirements engineering [Jørgensen and Lassen 2006; Machado et al. 2005].

Chapter 3

Software Requirements

Summary

This chapter introduces the requirements engineering area. The fundamental process followed in a requirements engineering project is detailed, including the description of models and actors involved in the process, and how it fits in the overall software engineering process. Afterwards, we briefly describe some approaches for requirements elicitation, which is concerned in defining where software requirements come from and how the software engineer can collect them. The chapter ends with a description of approaches for requirements validation, including some graphical animation techniques.

Contents

3.1 Introduction 38
3.2 Requirements Engineering Process 41
3.3 Requirements Elicitation and Analysis 43
3.4 Requirements Validation 50

3.1 Introduction

Requirements engineering is "the process by which the requirements for software systems are gathered, analyzed, documented, and managed throughout their complete lifecycle" [Aurum and Wohlin 2005]. Requirements engineering activities traditionally focus on the functional and non-functional aspects of the system to be developed, assuming that the organizational needs and context are outside from their competences.

The requirements for a system express the needs of the customers for a system that contribute to the solution of the real-world problem, thus in more general terms, a requirement is something the product to be delivered to the customer must do or a quality it should have. Software requirements are used to mean requirements for systems that are dealing with problems to be addressed by software artifacts. Software requirements are often either classified as functional or non-functional requirements:

- **Functional requirements** are things that the product should do, and they are constituted by statements of actions that the product must provide to the users, detailing how the system should react to particular inputs and how the system should behave for some situations. These requirements may also explicitly state what should not do.

- **Non-functional requirements** are properties (or qualities) that the product should have, and they are constituted by constraints on the actions offered by the system. Examples of constraints are timing constraints on the response of the system, and constraints on development process and standards.

The distinction between these two different types of requirements is not so clear, because there are examples of requirements that could be classified as both type of requirements, depending on the perspective. Sommerville [2006] suggests the existence of a third group called domain requirements, which are given by the application domain of the system and express characteristics of that domain, using either functional or non-functional requirements. Ideally, the specification of the functional requirements of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory definitions. In practice, for large, complex systems, it is practically impossible to have consistent and complete requirements.

Notice that the term "requirement" is used in different situations. There are cases where it is simply a high-level, abstract statement of a service that the system should provide or a constraint on the system. In other cases, it is used to refer to a detailed, formal definition of a system function. These two levels of detail of description are called as user requirements and system requirements, respectively. User requirements and system requirements are defined as follows [Sommerville 2006]:

- **User requirements** are constituted by high-level abstract requirements, and they are constituted by statements, in a natural language, complemented with diagrams, of what services the system is expected to provide and the constrains under which it must operate.

- **System requirements** are constituted by detailed descriptions of what the system should do, and they set out the system's functions, services and operational constraints in detail. The system requirements document should be precise, and should define exactly what is to be implemented. It can be used as a part of the contract between the system buyer and the software developers.

The not so clear separation between these different levels of description could create some problems during the requirements engineering process. For example, Abran et al. [2004, chap. 2] adopt the term system requirements to refer to the requirements for the system as a whole, that are the user requirements as described above.

The usage of different levels of system specification is useful because it facilitates the communication of information about the system being developed to different types of readers, who use them in different ways. For example, it is normal that a reader of the user requirements does not care about how the system is implemented. Examples of typical readers of user requirements include client managers, system end-users, client engineers, contractor managers, and system architects. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation. Examples of typical readers of systems requirements are system end-users, client engineers, system architects, and software developers.

Imprecision or ambiguity in the requirements specification are among the major causes of problems in software project. It is likely that software developers incorrectly handle an ambiguous requirement. However, this is

not what the customer wants. Consequently, when the mismatch is detected, the ambiguous requirements needs to be rewritten and changes must be introduced into the system. Obviously, this situation delays system delivery and increases development costs. For large and complex systems, it is easier to obtain a requirements specification that is imprecise, inconsistent or incomplete. One reason is that different system stakeholders may have different, and often conflicting, needs. These conflicting needs often give origin to inconsistent requirements that may not be simple to detect when the requirements are first specified. The problem may only emerge after deeper analysis or, sometimes, after development is complete and the system is delivered to the customer.

This chapter aims to introduce some of the topics on software requirements knowledge area as described in [Abran et al. 2004, chap. 2] (see Figure 3.1).

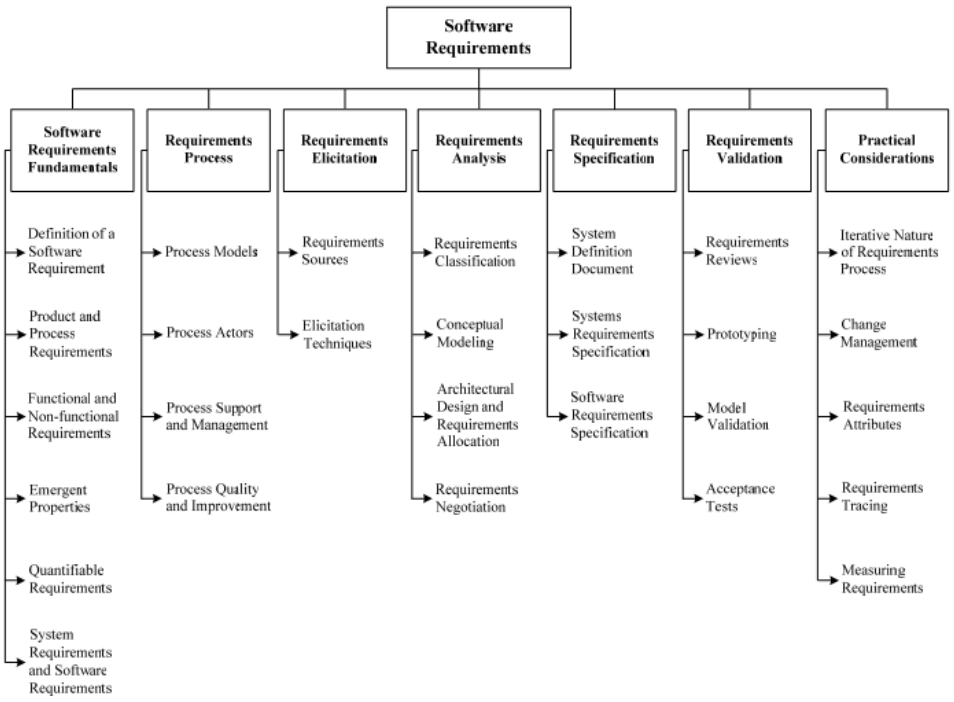


Figure 3.1: The topics for the software requirements (from [Abran et al. 2004]).

Software Requirements fundamentals are covered above in this section.

Section 3.2 presents the requirements process. Section 3.3 presents the requirements elicitation and the requirements analysis. Section 3.4 presents the requirements validation.

3.2 Requirements Engineering Process

A requirements engineering process aims to create and maintain a system requirements document. Roughly, there are three main activities inside a requirements engineering process:

1. In the **elicitation and analysis** activity, requirements are discovered through the observation of existing systems and the discussions with clients and potential users.

2. The **specification** activity converts the information gathered during the elicitation and analysis phase into a document that defines a set of requirements using an agreed format.

3. During **validation**, one checks if the requirements are precise, consistent, and complete. This activity tries to guarantee that the requirements actually define the system that the customer wants.

Inevitably, during the execution of these activities, some errors in the requirements document are detected. Consequently, the document must be changed in order to correct the errors. The requirements engineering process is concerned with the way the activities (elicitation and analysis, specification, and validation) are configured for different types of contexts. This process also includes activities that create artifacts that are needed during the requirements process, such as marketing and feasibility studies [Sommerville 2006].

One can observe that in almost all systems the requirements are subject to changes along the development process. Examples of factors that can impose changes to requirements are [Robertson and Robertson 2006]:

- The problem that the system is supposed to solve changes (economic or political reasons);

- The users change their minds about what they want the system to do, as they understand their needs better;

- The system environment changes;

- The new system is developed and released leading users to discover new requirements.

To cope with change, the requirements engineering activities should be organized as an iterative model, following for example the well-known spiral model [Boehm 1988]. One proposal that follows this idea is illustrated in Figure 3.2, which presents a process for the requirements engineering phase divided in the aforementioned three activities.

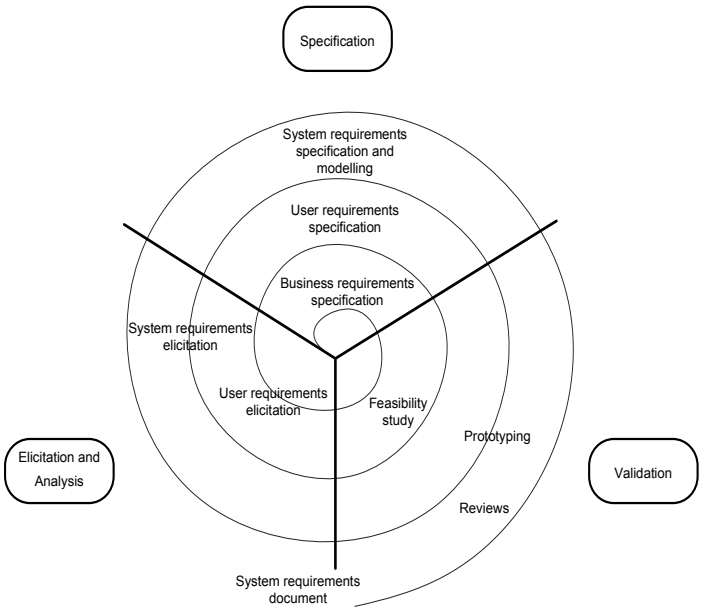


Figure 3.2: Spiral model for requirements engineering process (adapted from [Sommerville 2006, Chap. 7]).

The type of system under development and the stage of the overall process determine the time and the effort necessary to complete an iteration of each activity. When the process is in its initial stages the effort must be devoted on understanding high-level business, non-functional and user requirements. When the process is approaching the final stages the effort is spent on the system requirements engineering and system modelling.

The process of requirements engineering involves typically a group of people whose elements have different roles in the process. The term "stakeholder" is used, in requirements engineering, to refer to everyone who may influence or be influenced by the system. Thus, a stakeholder can be either a

client, a customer, an end-user, a project manager, an analyst, a developer, a senior manager or a member of the quality assurance staff. Each of these types of stakeholders have the following characterization based on what they expect from or gives to the software project being developed:

- The **client** is the one who pays the development of the product.
- The **customer** is the one who buys the software product, thus she aims to obtain the necessary functionalities at a lower price.
- The **end-user** is the one who ultimately interacts with the system, and she is interested in the introduction of useful functionalities that facilitate her work.
- The **project manager** wants to successfully complete the project with the given resources. She has knowledge about project management, software development and delivery process.
- The **system analyst** must correctly specify the requirements according to the stakeholders expections and needs. She knows about requirements methods and tools.
- The **system developer** aims to create on time and within budget a working software system. She knows about design methods, and programming environments and languages.
- The **quality expert** is responsible for ensuring that the software solution is compliant with quality standards.

Due to the diverse expectations and interests of stakeholders there are situations where it is necessary to negociate the requirements to let the collected requirement converge on a set of requirements that are mutually satisfactory for the stakeholders [Boehm et al. 2001].

3.3 Requirements Elicitation and Analysis

3.3.1 General considerations

In the activity of requirements elicitation, the requirements engineers work with customers and end-users to find out the application domain, the services the system should provide, the required performance of the system, the hardware constraints and so on. Requirements elicitation is considered to be among the most difficult, most critical, most error-prone, and most

communication-intensive aspects of software development [Wiegers 2003; Zowghi and Coulin 2005].

Eliciting and understanding stakeholder requirements is difficult for several reasons [Zowghi and Coulin 2005]. Often the stakeholders only describe in general terms what they expect from the software system, and they find difficult to describe in detail what they want from the system, and are not capable to evaluate the cost of their request and then they make unrealistic demands. The requirements engineers need to acquire experience in the domain of the customer, to understand the requirements given by the stakeholders who naturally tend to use the terms of their domain to express their requirements. Each stakeholder express the requirements in different ways. The situation implies that the requirements engineering must analyze requirement to discover commonalities and conflicts. Political factors may influence the requirements of the system. For example, managers may demand specific system requirements that will empower them within the organization. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. Hence, the importance of a particular requirement may change, and new requirements may emerge from stakeholders who were not originally consulted.

The elicitation and analysis activity is generally considered to include the following four tasks [Sommerville 2006]:

1. During the **requirements discovery** task, the requirement engineer interacts with stakeholders to collect their requirements;

2. The activity of **requirements classification and organization** takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters;

3. Inevitably, whenever multiple stakeholders are involved, requirements will most probably conflict. The **requirements prioritisation and negotiation** task is concerned with prioritizing requirements, and finding and resolving requirements conflicts through negotiation;

4. The **requirements documentation** task is concerned with the production of a document where all requirement are specified, either formally or informally.

As an example, let us detail the requirements discover task that typically includes steps that can be classified into five fundamental types, as described below:

1.a Analyzing the Stakeholders: One of the steps in requirements elicitation is to involve and to analyse all the relevant stakeholders, because they are the persons who have an interest in the system or are affected in some way by the development of the system.

1.b Understanding the application domain: The application domain [Zave and Jackson 1997] is the "real world" in which the system to be developed will ultimately reside. When the requirements elicitation starts, it is important to examine in detail the current environment of the application domain, in order to identify and describe the problems to be solved by the system according to the key business goals and issues.

1.c Identifying the Sources of Requirements: Stakeholders are the most obvious source of requirements for the system, and users and subject matter experts are asked to supply detailed information about the problems and user needs. When a project involves replacing a current or legacy system, the existing systems, including their documentation, and processes constitutes another source for eliciting requirements. It is important to identify the sources of requirements because requirements may be spread across many sources, in a variety of formats.

1.d Selecting the techniques, approaches, and tools to use: A critical factor in the success of the elicitation process is the correct choice of techniques or approaches to be employed in the specific context of the system to be developed [Nuseibeh and Easterbrook 2000], and also the tools to support it. It is generally agreed that there is no single elicitation technique or approach that is suitable for all projects.

1.e Eliciting the Requirements from Stakeholders and other Sources: After the identification of the requirements and the stakeholders, the requirements elicitation can start using the selected techniques, approaches, and tools.

The success of the requirements elicitation activity depends on the commitment and cooperation of the system stakeholders and on the communication skills of the requirements engineers. Thus, requirements elicitation can be seen as a multifaceted and iterative activity. The lack of agreement about the requirements of a system to be developed and the communication barriers among the stakeholders are two important problems that often occur in

software projects. This happens because it is typical to identify in a software development project different communities of participants, which imply that concepts clearly defined to one community could be entirely obscure for the members of another community.

3.3.2 Techniques and approaches for elicitation

In this subsection, some techniques and approaches for requirements elicitation are briefly introduced. In reality, there exist literally hundreds of techniques and approaches that can be employed for elicitation. Although not exhaustive, we consider our selection as representative of the most popular and applied techniques in industry. For a more detailed description of the techniques, please refer to [Zowghi and Coulin 2005].

Interviews [Holtzblatt and Beyer 1995] are essentially human-based social activities, and they are probably the most traditional and commonly used technique for requirements elicitation. The interviews are usually classified as structured, unstructured and semi-structured. Due the social nature of interviews, the quality of interaction between the participants determines the success of the activity, and the usefulness of the gathered information depends on the skill of the interviewers [Goguen and Linde 1993].

Questionnaires [Foddy 1993] consist in a list of questions using terms, concepts and boundaries of the domain previously established and understood by the participants. The questionnaires are mainly used during the early stages of requirements elicitation.

Task analysis [Richardson et al. 1998] constructs a hierarchy of the tasks performed by the users and the system, employing a top-down approach where high-level tasks are decomposed into subtasks and eventually detailed sequences until all actions and events are described.

Domain Analysis permits the analyst to gather early requirements and to understand and capture domain knowledge, by examining the existing and related documentation (e.g., forms and files used in the current business processes, and design documents and user manuals for existing systems) and also other applications. From these examined documents, some reusable concepts and components may be identified. Analogies and abstractions of existing problem domains can be used as baselines to acquire specific and detailed information, identify and describe possible solution systems, and assist in creating a common understanding between the analysts and the stakeholders. Domain knowledge in the form of detailed descriptions and examples plays an important part in the process of requirements elicitation, because it can be combined with other elicitation techniques.

These approaches also provide the opportunity to reuse specifications and to validate new requirements against other domain instances [Sutcliffe and Maiden 1998]. **Problem Frames** [Jackson 2000] permit one to detail the examination of problems allowing the identification of patterns that could constitute links to potential solutions.

Introspection [Goguen and Linde 1993] consists on developing requirements based on what the analyst believes the users and other stakeholders want and need from the system. This technique is applied when the analyst has a strong knowledge about the application domain and is aware of the goals of the system. In any case, it should be only seen as a starting point for using other requirements elicitation techniques

Repertory Grids [Jankowicz 2003] aim to identify and represent the differences and similarities between the different domain entities. They consist on building a matrix with the categories of the system, based on asking stakeholders to develop attributes and assign values to a set of domain entities. The level of abstraction present in the matrix is not familiar to most users, so it is usual to select this technique when eliciting requirements from domain experts. This technique is somewhat limited in its ability to express specific characteristics of complex systems.

When using **Card Sorting**, the stakeholders distribute a set of cards with the name of domain entities by groups, according to their own understanding, explaining the rationale for the way in which the cards are grouped. For effective card sorting, all entities must be included in the process. This is only possible when the domain is sufficiently understood by both the analyst and the participants. The information obtained using this technique is limited in its detail, due to the high level of abstraction represented by the entities.

Laddering [Fransella 2004] requires the stakeholders to answer short prompting questions, known as probes, which must be arranged into an organized structure. To be effective, the stakeholders must be able to express their understanding of the domain and then arrange it in a logical way. Like card sorting, laddering is mainly used as a way to clarify requirements and categorize domain entities. The resulting knowledge, which is typically displayed using a tree diagram, is reviewed and iteratively modified as more information is added.

Group Work is one of the most used techniques in industry for requirements elicitation. Groups are particularly effective because they involve and commit the stakeholders directly and promote cooperation. Due to the number of different stakeholders that may be involved in a given project, these sessions can be difficult to organise. The success of group work depend on

the participants and the cohesion within the group. Group work is less effective in highly political situations, because stakeholders must feel comfortable and confident in speaking.

Brainstorming is based in an informal discussion avoiding exploring or criticising ideas in great detail, to rapidly generate as many ideas as possible. Typically, brainstorming is used to find the preliminary mission statement for the project and target system. This technique allows the discovery of new and innovative solutions to existing problems.

Joint Application Development (JAD) [Wood and Silver 1995] investigates through a general discussion amongst all the available stakeholders both the problems to be solved, and the available solutions to those problems. It is usual that the main goals of the system have already been established before the stakeholders participate. The focus of this type of meeting tends often to be on the needs and desires of the business and users rather than technical issues. **Requirements Workshops** [Gottesdiener 2002] constitute a similar concept and are basically a group meeting, where the emphasis is on developing and discovering requirements for a software systems.

Ethnography [Ball and Ormerod 2000; Jirotka and Goguen 1994] is an elicitation technique that suggests the involvement of the analysts in the normal activities of the users over a significant period of time. This situation permits the analyst to collect information about the operations being executed, which is especially useful when dealing with usability factors, or studying interactions between human stakeholders. A special type of ethnography is **Observation**, where the analyst, as the designation suggests, observes (without direct interference) the execution of the existing processes by the users. This technique is commonly used in conjunction with interviews and task analysis.

In **Protocol Analysis** [Goguen and Linde 1993] the participants talk through an activity or task, describing the actions being conducted and the process behind them. Often, some minor steps performed frequently and repetitively are taken for granted by the users, and may not be explained and subsequently recorded as part of the process.

Apprenticing [Robertson and Robertson 2006] considers that an experienced user guides the analyst actually learning and performing the current tasks, thus the analyst becomes actively involved in the real life activities of the business.

Prototyping [Gomaa and Scott 1981] consists on building a prototype (i.e., a rudimentary version of the final system) based on the, usually preliminary, release of the requirements document, or the existing examples of

similar systems. By playing with the prototype, the stakeholders can get a feel of the system, which can be used to support the investigation of possible solutions to gather detailed information and relevant feedback [Sommerville 2006]. This technique is helpful when developing new systems for entirely new applications, and the prototypes encourage stakeholders to participate in elicitation of the requirements. In many cases, prototypes are expensive to produce in terms of time and cost. Prototypes are normally used in conjunction with other elicitation techniques, such as interviews and JAD.

Goal-based approaches consider that high-level goals representing the objectives for the system are decomposed and elaborated into sub-goals. This decomposition approach can be conducted until the individual (low-level) requirements are elicited. As the result of this process, one obtains a detailed relationship between the objectives of the system, the requirements and the domain entities. The decomposition of objectives of the system is usually done using AND and OR relationships, and the elaboration is usually done using two kinds questions, one about reason why it is done, and another about how it will be done. The possibility to introduce errors when defining the high-level goals of the system constitute one of the main risks when using goal based approaches, because it can have a major and damaging effects in the future phases of the development, and the changing of goals during the development are difficult to manage. Significant effort has been devoted to developing these types of approaches for requirements elicitation such as the F^3 *project* [Bubenko and Wangler 1993], the *KAOS meta model* [Dardenne et al. 1993] and the *i* framework* [Yu 1997]. The use of goals in conjunction with scenarios to elicit requirements has also attracted considerable attention [Haumer et al. 1998; Potts et al. 1994; Rolland et al. 1998]. In practice, all these goal-based approaches have been applied in contexts where only the high-level needs for the system are well-known, and there exists a general lack of understanding about the specific details of the problem to be solved and its possible solutions.

Scenarios are basically descriptions of actions and interactions between the users and the system. The descriptions in the scenarios capture examples and illustrations, thus it helps stakeholders in reasoning about complex systems [Potts et al. 1994].

Scenarios are widely used in requirements elicitation and are narrative and specific descriptions of current and future processes, including actions and interactions between the users and the system. Like use cases, scenarios do not typically consider the internal structure of the system, and require an incremental and interactive approach to their development. Naturally, it is important when using scenarios to collect all the potential ex-

ceptions for each step. A substantial amount of work from both the research and practice communities has been dedicated to developing structured and rigorous approaches to requirements elicitation using scenarios, including CREWS (Cooperative Requirements Engineering With Scenarios) [Maiden 1998], The Inquiry Cycle [Potts et al. 1994], SBRE [Kaufman et al. 1989], and Scenario Plus [Scenario Plus 2008]. Scenarios are additionally very useful for understanding and validating requirements, as well as test case development.

Scenarios can be used to describe possible instantiations of a given use case. Each scenario describes a specific sequence of actions and interactions between the users and the system. Among other alternatives, scenarios can be expressed either as a list of steps written in natural language, or by a UML interaction diagrams. It is important to notice that scenarios only provide restricted requirements descriptions, because they deal with examples and illustrations. Therefore, the scenarios must be generalised to obtain the complete requirements.

Viewpoint approaches allow the modelling from different perspectives of the domain in order to develop a complete and consistent description of the system. For example, a system can be described in terms of its operation, implementation and interfaces. In the same way systems can be modelled from the standpoints of different users or from the position of related systems. These types of approaches are particularly effective for projects where the system entities have detailed and complicated relationships with each other. Viewpoints are also useful as a way of supporting the organization and prioritization of requirements. However, they are usually criticised due to the fact that they do not enable non-functional requirements to be represented easily, and are expensive to use in terms of the effort required. Some viewpoint approaches [Nuseibeh et al. 1996; Sommerville et al. 1998] provide a flexible multi-perspective model for systems, using different viewpoints to elicit and arrange requirements from a number of sources. Using these approaches analysts and stakeholders are able to organize the process and derive detailed requirements for a complete system from multiple project specific viewpoints.

3.4 Requirements Validation

This section discusses issues related with the requirements validation activity, which occurs during the requirements engineering process. Requirements validation is based on the elaboration of a demonstration that the gathered

requirements define the system that the customer really wants. It is important to clarify the distinction between validation and verification of a model, since in some contexts these two terms are used interchangeably. **Validation** is the process of determining the degree to which a model is an accurate representation of the real-world from the perspective of the intended uses. **Verification** is the process of determining that a computer model accurately represents the developer's conceptual description and specification. These terms have a different interpretation, for example, in the simulation area [Law and Kelton 2000], where verification consists on ensuring that the model behaves as its creators intended, while validation consists on ensuring that the model has a similar behaviour to the real system.

The importance of the requirements validation activity is related to the higher cost of fixing an error at the design or implementation phases of a project when compared with the correction at the requirements phase. Similarly to the requirements analysis, the requirements validation activity is focused on finding problems, omissions, and mismatches at the requirements. To accomplish this purpose, the validation activity includes different types of checks to the requirements document [Sommerville 2006]:

- **Validity** aims to check if the system provides the functions which best support the customer's needs. Typically, in the beginning of the process the user thinks that a system is needed to provide certain functions. However, along the process with more thought and analysis of the needs one may identify additional or different functions that are required to be available in the final product. As we said before, a system have different types of stakeholders who have distinct needs. In this way, any set of requirements is inevitably a compromise among the different opinions and expectations of the various stakeholders.

- **Consistency** aims to identify if there are any conflicts among the requirements. It is expected that the requirements expressed in the document have no conflicts, that is, there should be no contradictory constraints or descriptions of the same system function.

- **Completeness** aims to check if all functions required by the customer are included. The requirements document must include the requirements, which define all the functions and all the constraints expressed by the system user.

- **Realism** aims to verify if the requirements can be implemented given the available budget and technologies. The requirements should be

checked to ensure that they could be implemented using the competences and skills on the existing technologies, and taking into account the budget and schedule available for the system development.

- **Checkability** aims to answer to the question *"Can the requirements be checked?"*. Typically, there is a potential dispute between the customers and client that can be reduced by assuring that the system requirements are all described in such a way that they can be checked. This must allow the specification of a set of tests to ensure that the delivered system meets each specified requirement.

There exist several techniques for requirements validation. Some of them can be used in conjunction in some situations. Those techniques can be classified in the following three main types:

1. The **requirements reviews** is based on the systematic and manual analysis of the requirements document by a team of reviewers.

2. The **prototyping** creates an rudimentary version of the system under development to be used to demonstrate concepts and try out design options.

3. The **test-case generation** develops tests to check the requirements, from data, logical relationships or other requirements information. A common practice of eXtreme Programming (XP) is to develop tests from the user requirements before writing any code.

A requirements review is a manual task to check the requirements document for anomalies and omissions, which whenever possible should involve a number of stakeholders with different backgrounds. A possible form of requirements review consists on the development team driving the client over the system requirements, explaining the impact of each one. The review team should check each requirement for consistency, as well as checking the set of requirements as a whole for completeness.

All detected conflicts, contradictions, errors and omissions in the requirements should be pointed out by the reviewers and formally recorded in the review report. This report is an useful artefact for the stakeholders, when they have to participate in the (re)negotiation of the requirements.

It is surprising how often communication between system developers and stakeholders ends after the elicitation activities. Often, there is no confirmation that the documented requirements reflect what the stakeholders really said (or at least thought) about what they want the system to provide. In

many cases, problems can be detected simply by talking about the system with the stakeholders.

There are many distinct situations in software development projects where is may be useful to apply the prototyping technique. One example is when the prototype constitutes an experimental (rudimentary) system, specifically developed to elicit the requirements. Therefore, it is common to start with requirements that are poorly understood, and progressively enrich and improve the prototype, along the requirements engineering process, as requirements are better verbalised and documented. Another example is when the project involves multiple teams working on different places. In this situation, it is important to deliver a prototype to the end-users, i.e., a working system created based on the currently available set of requirements, to validate the agreed and negotiated requirements. When the requirements are validated, in some contexts, the prototype is thrown away and the design and implementation activities can begin, using different approaches and technologies (than those adopted for the prototype).

Agreeing on the requirements for a given system is not at all easy or simple, since it includes many problems. Among those problems we can identify difficulties, for example, on the communication amongst the stakeholders and on the combination of several different opinions and perspectives when multiple stakeholders are involved. Therefore, it is generally accepted that it is difficult to show that a set of requirements meet the needs of the users, so efforts must be done in order to allow them to form a mental picture of the system and imagine how that system fits into their work [Sommerville 2006]. If the prototype is an executable model, it can be used to validate requirements by executing it, so that end-users and customers can appreciate the intended behaviour of the system.

To make effective the validation activity, the execution or simulation of the considered model must be easily followed by the end-users. **Simulation** refers to a large set of methods that intend to mimic the behaviour of real systems, facilities, and processes, usually on a computer with adequate software [Law and Kelton 2000]. Simulation is one of the most widely used operations-research and management-science techniques, with numerous and diverse application areas.

Typically, the model being simulated follows a specific formal modeling language, in which end-users and customers are not experts. Thus, it is highly desirable to provide a visualization of the simulation of the model (and consequently of the system), using representations of the elements of the application domain. Since end-users and customers are familiar with the application domain and its elements, it is expected that they will not

have major cognitive problems in understanding the visualisation, if the representations are sufficiently intuitive.

In this thesis, we propose to use animations as a form of visualising the execution/simulation of a model of the system under consideration. **Animation** is the rapid display of a sequence of images of two-dimensional or three-dimensional artwork or model positions, in order to create an illusion of movement. The most common method of presenting animation is as a motion picture or video program, although several other forms of presenting animation also exist. An animation can be used to visualise the model behaviour using graphics from the application domain. An emblematic example of a tool that allows the animation of models is the ARENA tool [Kelton et al. 2004].

An animation represents the key elements of system on the screen by icons that dynamically change their appearance (colour and shape) and their position as the simulation model evolves through time. Some of the usages of an animation are:

- To communicate the essence of a simulation to a manager or to other people who may not be aware of (or care about) the technical details of the model;
- To debug the simulation models;
- To show that a simulation model is not valid;
- To suggest improved operational procedures for a system, because in many situations some phenomena may not be apparent just by looking at the simulation's numerical results;
- To prepare or to train the people who will use the future system;
- To improve the communication among the project team.

There are two main types of executing an animation. In the first one, called concurrent animation, the animation is being displayed at the same time that the simulation is running. One must be aware that in these cases the animation may significantly slow down the execution of the simulation. In the second type, designated post-processed animation (also called playback), the state changes in the simulation are recorded and used to drive the graphics, after the simulation is over.

An animation is primarily a communications method, so it should be possible to create high-resolution icons and to save them for later reuse. A

library with standard icons should be available in the animation tool, and it must allow a smooth movement of icons. It is desirable if animation uses vector-based graphics (pictures are drawn with lines, arcs and fills) rather than pixel-based graphics (pictures are drawn by turning individual pixels on or off). The former type of graphics allows rotation of an object (e.g., a helicopter rotor). The animation tool must also offer a set of commands to control the animation, such as speeding up or slowing down the animation, zooming in or zooming out parts of the system, and panning to see parts of the system that are too large to fit in the screen. Some software products have named animation views, so that one can construct a menu of views corresponding to different parts of the simulated system.

Part II

Contribution

Chapter 4

Transforming Sequence Diagrams into a CPN Model

Summary

This chapter introduces a set of rules to transform a set of sequence diagrams of UML 2.0 into a CPN model. It is detailed how to transform plain sequence diagrams, which define the messages in the sequence diagrams and the order between them, in terms of constructs of the CPN modelling language. Afterwards, one defines how some of the high-level operators of the sequence diagrams are transformed into elements of the CPN model.Additionally, are shown the results of applying these ideas to some illustrative examples. The tool support available for the rules to transform sequence diagrams into a CPN model is described.

Contents

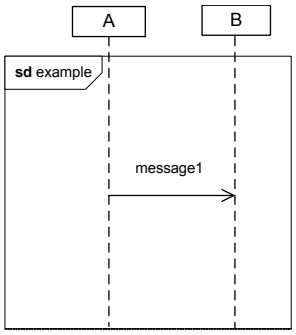
4.1 Plain Sequence Diagrams 60
4.2 Sequence Diagrams with High-level Operators . 62
4.3 Tool Support 67

4.1 Plain Sequence Diagrams

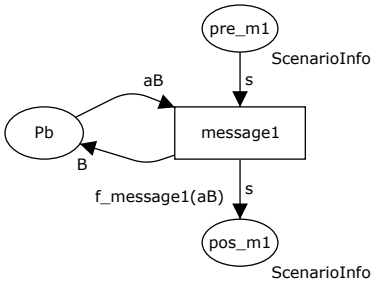
This section illustrates how to transform plain sequence diagrams into a CPN model. Plain sequence diagrams refers to UML 2.0 sequence diagrams that have no high-level operators. To accomplish this, we describe how the transformation is achieved, depending on the semantics of the operators, and additionally we show the result of applying these ideas to some illustrative examples. The material presented in this chapter was published in [Ribeiro and Fernandes 2006].

We consider a semantic for sequence diagram with a order relation between messages such that the emission of a given message requires the reception of the preceding one, if it exists. Notice that we are assuming sequence diagrams with synchronous messages.

Firstly, let us consider the case of a sequence diagram with only one message. Figure 4.1 contains both the representation of a sequence diagram with only one message (Figure 4.1a), and the corresponding CPN model (Figure 4.1b). The message called `message1` flows from the lifeline of object `A` to the lifeline of object `B`.



(a) A sequence diagram with only one message (message1).



(b) The obtained CPN model.

Figure 4.1: Transforming a sequence diagram with only one message.

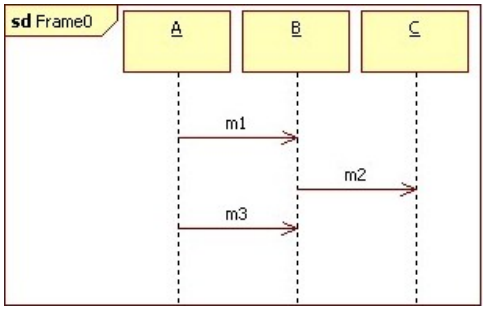
Each message in a sequence diagram is represented in the CPN model by a transition, thus the firing of a transition in the CPN model represents the execution of the corresponding message in the sequence diagram. The CPN model presented in Figure 4.1b has the transition `message1` that represents the message in the (source) sequence diagram.

The place `pre_m1` represents the initial step of execution of the sequence diagram, that is tokens in this place allow the execution of the part of

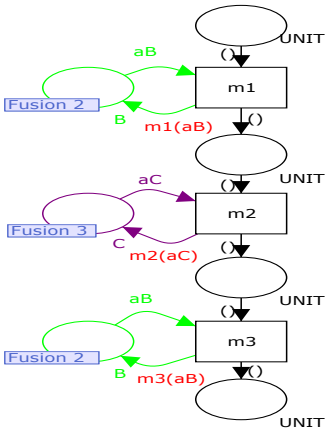
the CPN model that represents the body of the sequence diagram. In this example, the body of the sequence diagram is the execution of `message1`. The place `pos_m1` represents the last step of the execution of the sequence diagram.

There are places in the CPN model that hold the current values of the object that appears in the sequence diagram. Each message in a sequence diagram carries a function that may change the value of the receptor object. In this way, each time a transition fires, the values of objects may change. Next to each transition in the CPN model there is a place that holds a representation of the current value of the object in the message's destination. In Figure 4.1b, place `pb` that holds the tokens of the current values of instances of object `B`.

The sequence diagram in Figure 4.2a represents an interaction without high-level operators. There are three objects (`A`, `B` and `C`) with the corresponding lifelines. There exist three messages in the sequence diagram:



(a) A UML 2.0 sequence diagram without high-level operators



(b) The obtained CPN model

Figure 4.2: Example of transform a sequence diagram without high-level operators

1. message `m1`, that flows from `A` to `B`;
2. message `m2`, that flows from `B` to `C`;
3. message `m3`, that flows from `A` to `B`.

One can observe that message `m1` appears before message `m2`; and message `m2` appears before `m3`. Figure 4.2b depicts the CPN model obtained from the sequence diagram. The messages `m1`, `m2` and `m3`, are represented by the transitions with the same names in the CPN model. When executing the CPN model the firing of a transition represents the execution of the corresponding message in the sequence diagram.

The order between the messages is ensured by introducing places between the transitions with the colour set `UNIT`. The input place for transition `m1` represents the beginning of the execution, the output place for transition `m3` represents the end of the execution, and the places between two transitions (i.e. a place that is input place for one transition and output place for another) guarantees the order between the firing of transitions.

An *InteractionFragment* is a set of *Lifelines*, each of which has a sequence of *EventOccurences* associated with it.

4.2 Sequence Diagrams with High-level Operators

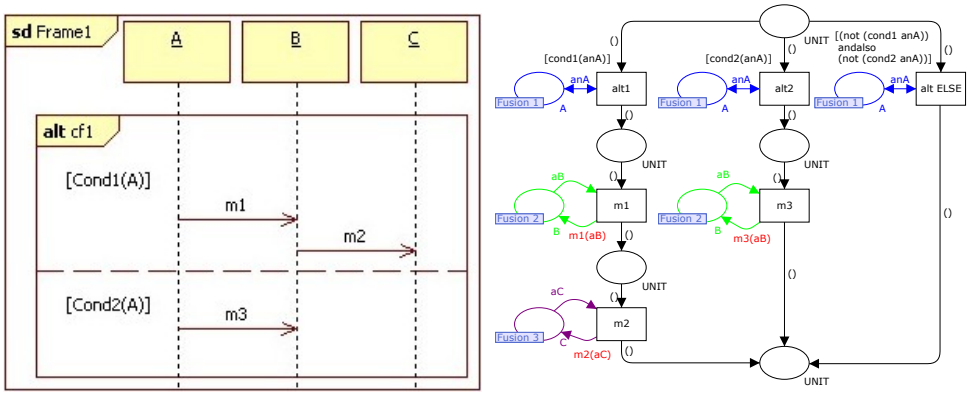
This section illustrates how to translate some of the high-level operators available in the UML 2.0 sequence diagrams, into a behaviourally-equivalent CPN model. We restrict our study to the following set of high-level operators: `strict`, `seq`, `par`, `loop` and `alt`. Operators like `neg`, `assert`, and `critical` are not considered in this work. The operators present in a sequence diagram one described in Section 2.2.2.

4.2.1 Alternative choice

The choice of behaviour is represented by a *CombinedFragment* with the *interactionOperator* `alt`. Each operand of `alt` has an associated guard, which is evaluated when choosing the operand to be executed. No more than one operand will be chosen and in this work we assume that the guards must be mutually disjoint (exclusive). When one of the operands has its guard evaluated to true, the interaction associated with this operand is considered. The empty guard is by default evaluated to true. The operand guarded by `else` is evaluated to true only when the guards in the other operands are all evaluated to false. In the case that none of the operands' guards are evaluated to true (this means that there are no `else` and empty guards), operand is executed.

The sequence diagram in Figure 4.3a is transformed into the CPN model in Figure 4.3b. Each operand in the sequence diagram is transformed into

a sequential branch. All sequential branchs begin in a common input place and end in a common output place.



(a) A UML 2.0 sequence diagram (b) The obtained CPN

Figure 4.3: Example with the alternative choice operator (*alt*)

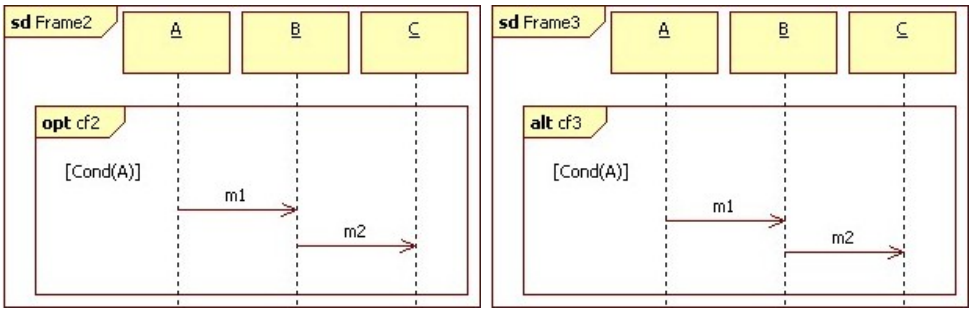
Please notice that in this case there is no `else` guard, and thus when none of the guards is evaluated as true, no operand is executed. In terms of the CPN model, this is represented by its rightmost branch where the condition of the "`alt ELSE`" transition is the negated disjunction of all other guards. The other branches are guarded by the same condition as in the sequence diagram and describe the same sequence.

4.2.2 Optional

The optional operator, represented by *InteractionOperator* `opt`, can be seen as an alternative choice with only one *Operand*, whose guard is not the `else` (see Figure 4.4). With this similarity, we can apply to the optional operator the same general translation scheme used for alternative choice.

4.2.3 Parallel composition

The parallel merge between two or more behaviours is represented by a *CombinedFragment* with the *interactionOperator* `par`. Keeping the order imposed within each operand, *EventOccurrences* from different operands can be interleaved in any way. The sequence diagram in Figure 4.5a is transformed into the CPN in Figure 4.5b. The obtained CPN model has two additional transitions to control the interleaving of behaviours. The transi-

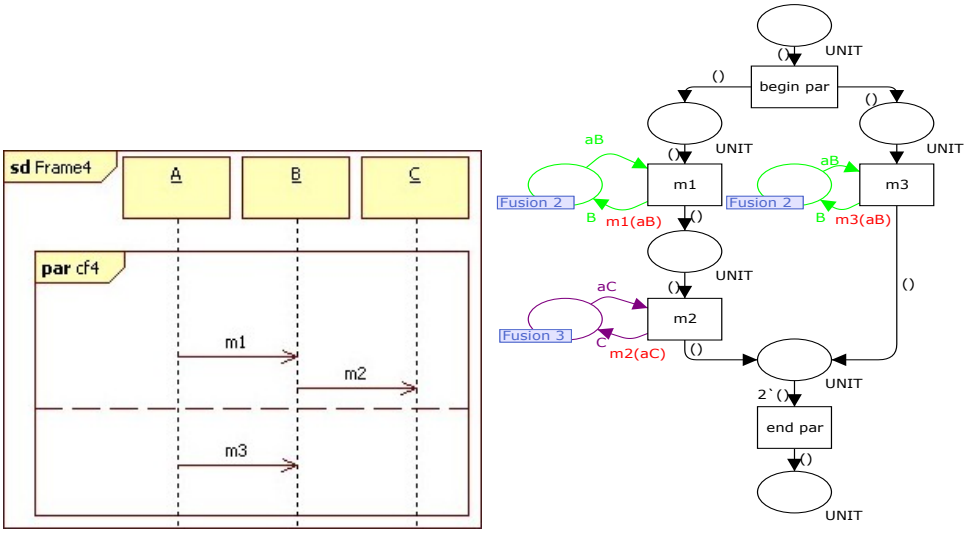


(a) A UML 2.0 sequence diagram

(b) The corresponding UML 2.0 sequence diagram with `alt`

Figure 4.4: The option operator (*opt*) expressed by an alternative choice.

tion "`begin par`" creates two branches (one for each operand) introducing a token into the two output places. This way, we obtain the interleaving between the transitions of each branch. The transition "`end par`" waits for the execution of all created branches, because it is enabled only when its input place has a number of token equal to the number of created branches.



(a) A UML 2.0 sequence diagram

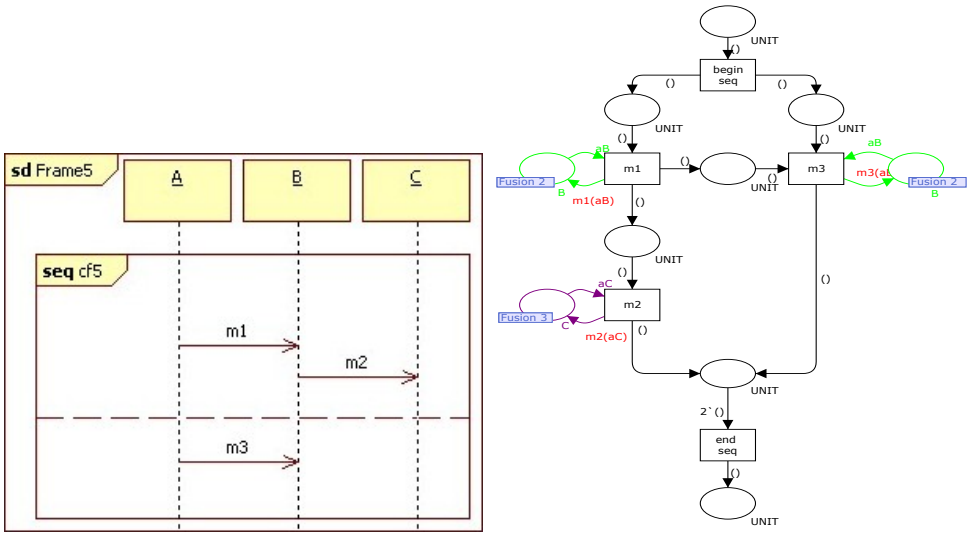
(b) The obtained CPN

Figure 4.5: Example with the parallel composition operator (*par*)

4.2.4 Weak sequencing

When using the *InteractionOperator* `seq` the corresponding *CombinedFragment* represents a weak sequencing between the behaviours of the operands. The ordering of *EventOccurrences* within each of the operands are maintained in the result. *OccurrenceSpecifications* on different lifelines from different operands may come in any order. *OccurrenceSpecifications* on the same lifeline from different operands are ordered such that an *EventOccurrence* of the first operand comes before that in the second operand.

In Figure 4.6a we have an example of a sequence diagram with `seq` operator. The messages `m1` and `m3` have the *EventOccurence* in the same *Lifeline*, and in the first operand, after the message `m1` we have the message `m2`. Thus, message `m1` must occur before messages `m3` and `m2`.



(a) A UML 2.0 sequence diagram

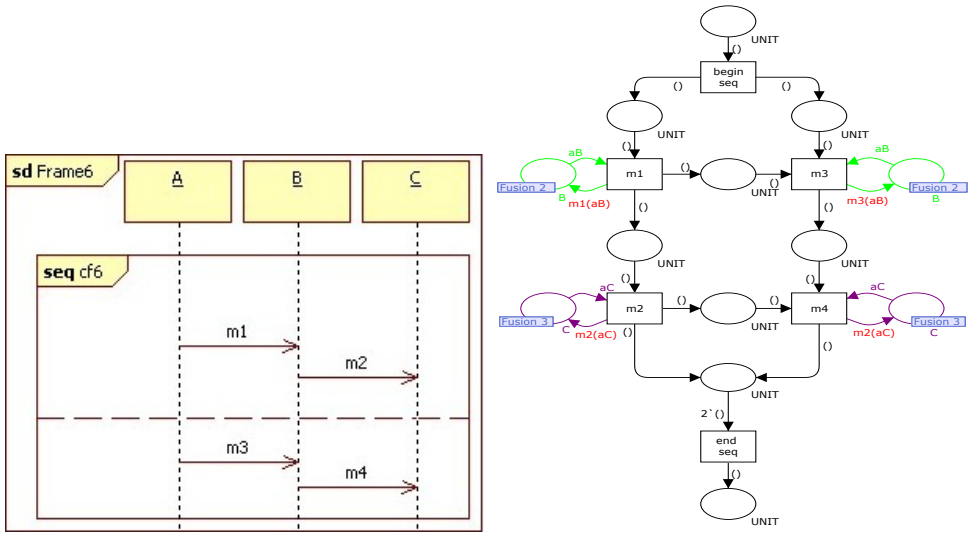
(b) The obtained CPN

Figure 4.6: Example with the weak sequencing operator (*seq*)

To construct from a sequence diagram with the `seq` operator the corresponding CPN model, we first consider the structure for the parallel composition between the operands, and after that we impose some more order between transitions in different branches.

The sequence diagram in Figure 4.7a is another example using the operator `seq`. The corresponding CPN model is presented in Figure 4.7b.

There are some particular cases using this operator. If the *EventOccur-*



(a) A UML 2.0 sequence diagram

(b) The obtained CPN

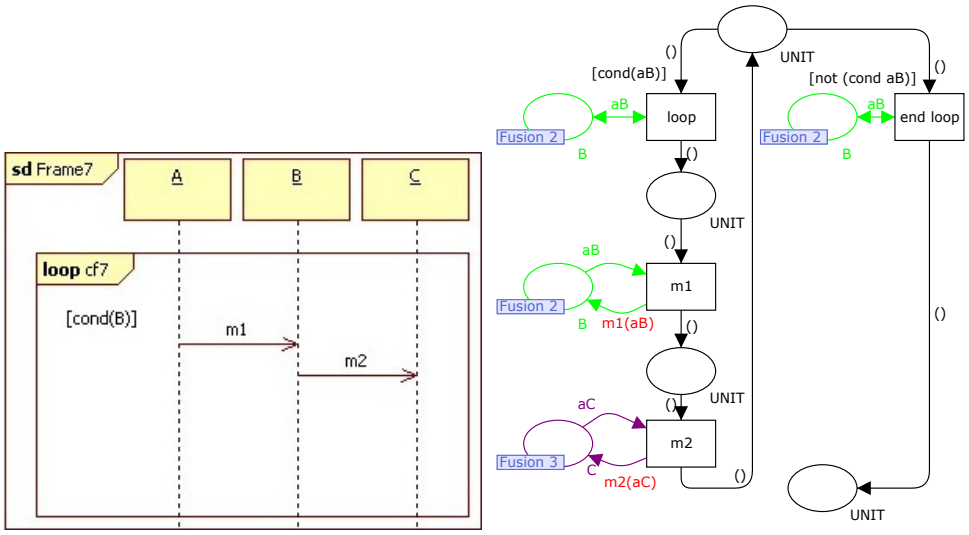
Figure 4.7: Another example with the weak sequencing operator (*seq*)

rence of the last message from the first operand is in the same *Lifeline* as the first message of the second operand, we have a sequential order between all the messages in the operands. If none *EventOccurrence* of messages is in the same lifeline we have a parallel composition between the operands.

4.2.5 Looping

The `loop` *InteractionOperator* represents the iterative application of the operand in the *CombinedFragment*. This iterative application can be controlled by a guard or by a minimum and maximum number of iterations.

Given the CPN module for the operand inside the `loop`, we add two transitions: "`loop`" and "`end loop`". These two transitions have the same input place. Transition "`loop`" is enabled if the condition (guard for `loop` operator) evaluates to true, and its output place is the input place for the operand's CPN. The transition "`end loop`" is enabled when the condition evaluates to false and its output place is used as connection to the end of the `loop` operator. In Figure 4.8 we have an example of a sequence diagram with the `loop` operator.



(a) A UML 2.0 sequence diagram (b) The obtained CPN

Figure 4.8: Example with the looping operator (*loop*).

4.3 Tool Support

This section details a prototype of a tool that permits one to transform in an automatic way a set of sequence diagrams describing scenarios of a given system, into a CPN model. We have constructed our tool following a model-to-model transformation approach [Czarnecki and Helsen 2006]. Model transformations are described by means of the so called transformation languages [Sendall and Kozaczynski 2003]. The tool uses the mechanisms provided by the Atlas Transformation Language (ATL) [Jouault et al. 2008], namely to implement the transformation. ATL is supported by a set of development tools built on top of the Eclipse environment [Bézivin et al. 2004].

We adopt this approach in order to reuse the existing tools for the involved metamodels, in particular the tools to manipulate the sequence diagrams have the capability of exporting the diagrams using a XML standard language.

4.3.1 Description of the involved metamodels

As explained before, the behaviour of each use case is described by a collection of UML 2.0 sequence diagrams, which constitute the source metamodel for the transformation considered here. UML 2.0 has been launched to substitute the previous versions, and among other changes it introduces in sequence diagrams many new high-level flow operators. The description of the metamodel for UML 2.0 is presented in Section 2.2.

In this work we are assuming a linear semantics for the sequence diagrams, which considers basically that if a message *m1* precedes a message *m2*, then *m1* must be received before the sending of *m2*. According to [Sibertin-Blanc et al. 2008], the usage of this linear semantics is appropriate when considering sequence diagrams that capture the behaviour given by the interactions among the actors and the system.

The ATL transformation engine accepts XMI serialisations of models and metamodels that conform to the Meta-Object Facility (MOF), which is the metamodeling architecture to describe models introduced by the OMG. The ECORE framework can be seen as a practical implementation of MOF. We consider an implementation of the UML2 metamodel defined in the ECORE framework and provided by the UML2 Eclipse project [Eclipse UML2 2008]. Thus, it is possible to use a UML modelling tool that exports to XMI in order to create the UML2 sequence diagrams.

An Example of a Sequence Diagram Model

Let us see how the sequence diagram in Figure 4.9 is represented in the UML2 metamodel provided by the Eclipse project.

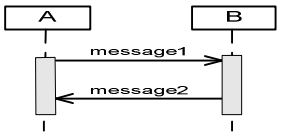


Figure 4.9: An example of UML2 sequence diagram.

Figure 4.10 shows an example on how sequence diagrams are edited in the UML editor of the Eclipse modelling tools, where the elements of the model

are shown in an hierarchical, tree-like view. Each element of the sequence diagram is preceded by a classifier (and also an icon associated with it) that represents the element in the metamodel. This view of the sequence diagram shows its elements and allows the navigation on the elements. The properties associated to each element can be found in a separate tab.

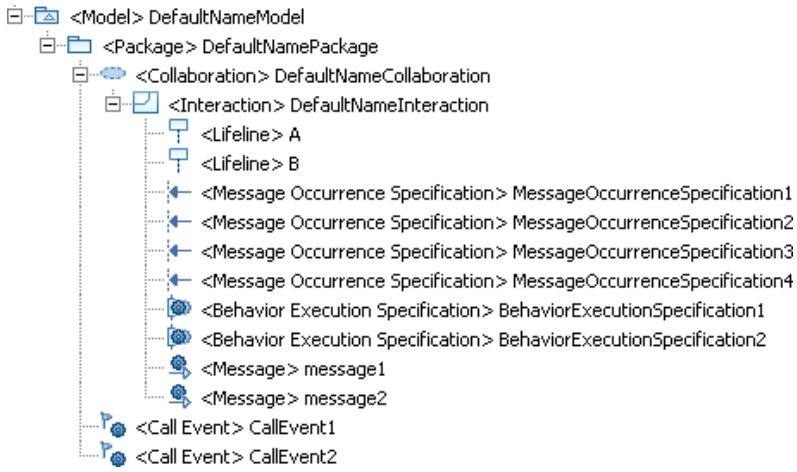


Figure 4.10: The elements of the sequence diagram in Figure 4.9 shown in the UML editor of the Eclipse modelling tool.

Figure 4.11 presents fragments of the XMI code that represents the sequence diagram in Figure 4.9. Lines 1 and 2 present the code for the lifeline corresponding to the `A` entity, with the corresponding list of elements in it, given by the attribute `coveredBy`. Lines 4-7 present the code for the message occurrence specification `MOS(1)`, that is one of the elements in the lifeline of the entity `A`. Similarly, lines 9-12 show the code for the message occurrence specification `MOS(4)`, which is another element in the lifeline of the entity `A`. Lines 14-16 detail the behaviour execution specification present in the lifeline of the entity `A`, that starts with `MOS(1)` and finishes with `MOS(4)`.

Lines 18-19 describe the message called `Message1`, detailing its send event, that is the message occurrence specification `MOS(1)` (present at the lifeline of the entity `A`); and its receiving event that is the message occurrence specification `MOS(2)` (present at the lifeline of the entity `B`).

```
1  <lifeline xmi:id="LifeLine(A)" name="A"
2          coveredBy="MOS(1) MOS(4) BES(1)"/>
3  ...
4  <fragment xmi:type="uml:MessageOccurrenceSpecification"
5          xmi:id="MOS(1)" name="MessageOccurrenceSpecification1"
6          covered="LifeLine(A)" event="CallEv(1)"
7          message="Msg(1)"/>
8  ...
9  <fragment xmi:type="uml:MessageOccurrenceSpecification"
10         xmi:id="MOS(4)" name="MessageOccurrenceSpecification4"
11         covered="LifeLine(A)" event="CallEv(2)"
12         message="Msg(2)"/>
13 ...
14 <fragment xmi:type="uml:BehaviorExecutionSpecification"
15         xmi:id="BES(1)" name="BehaviorExecutionSpecification1"
16         covered="LifeLine(A)" start="MOS(1)" finish="MOS(4)"/>
17 ...
18 <message xmi:id="Msg(1)" name="Message1"
19         receiveEvent="MOS(2)" sendEvent="MOS(1)"/>
```

Figure 4.11: XMI code corresponding to the sequence diagram presented in Figure 4.9.

A metamodel for CPN

The CPN modelling language is the target metamodel for the transformation. The CPN Tools saves the CPN models being edited in a XML-based format, whose description can be found in [CPN Tools 2009]. Based on this description, we have created a description, using the UML notation, of a part of the metamodel used in *CPN Tools*. Figure 4.12 shows the UML description of the metamodel for CPN models, which is used to support the process of obtaining a CPN model, within the model transformation approach considered in this work.

The CPN metamodel presented in Figure 4.12 has the element `CPN` as its main element. Each `CPN` has a non-empty set of modules (also called pages), which are represented by the element `Page`. Each `Page` can have two kinds of artefacts: the nodes and the arcs. The `Node` element is defined as an abstract class (name is presented in italic), and it can be used to represent either a `Transition` element, or a `Place` element.

The element `Arc` is associated with one place and one transition, and each `Arc` has an orientation and an inscription. The possible orientations of an `Arc` is defined as the enumeration given by `ArcKind`, where `PtoT` represents an arc that goes from a place to a transition, `TtoP` represents an arc that

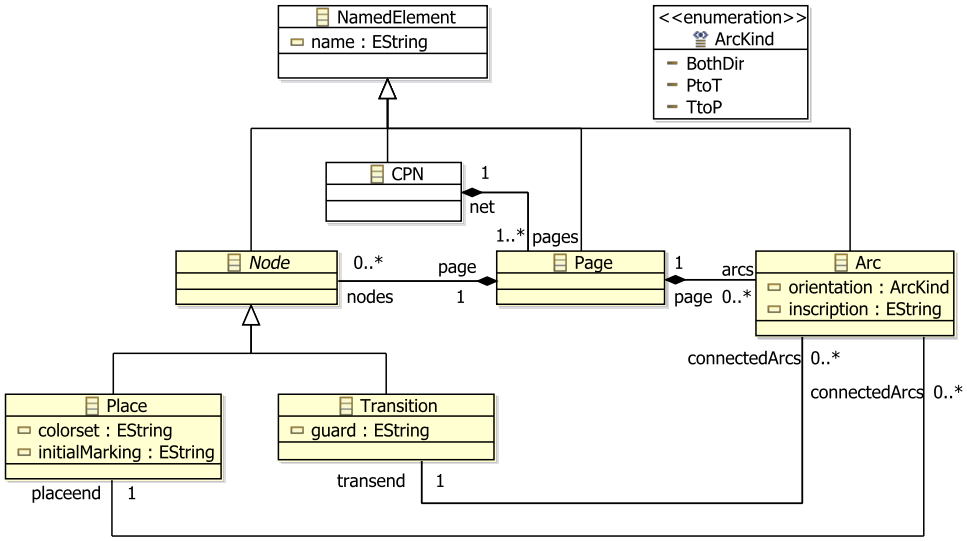


Figure 4.12: Metamodel of CPN modelling language.

goes from a transition to a place, and `BothDir` represents a bidirectional arc between a place and a transition. The arc inscriptions are represented as a `String` data type.

This metamodel does not address the graphical properties of the elements (like places, transitions or arcs) of a CPN model. We use an additional transformation of a CPN model into a textual representation of the corresponding XML-file to allow its usage within the CPN Tools. Default values for the graphical properties of each element are used when generating the XML representation of a CPN model. We are assuming that the user can use the tool facilities to rearrange the graphical layout of the CPN model.

CPN models following the metamodel in Figure 4.12 can be represented in XMI code. Figure 4.13 presents part of the XMI file to represent the CPN model in Figure 2.8.

In Figure 4.13, lines 1-3 describe the transition `T1`, that is a node of type `Transition` with the name `T1`, and the arcs that are connected with the node, which are determined by a set of references to parts of the XMI document. For example, one of the arcs connected with `T1` is the arc referred as `arcs.0`, which means the first occurrence of the tag `arcs` in the document.

```
1  <nodes xsi:type="CPN:Transition" name="T1"
2         connectedArcs="//@pages.0/@arcs.0 //@pages.0/@arcs.1
3                        //@pages.0/@arcs.2"/>
4  ...
5  <nodes xsi:type="CPN:Place" name="P1"
6         colorset="Int" initialMarking="1`1++1`5"
7         connectedArcs="//@pages.0/@arcs.0 //@pages.0/@arcs.5"/>
8  ...
9  <arcs name="fromP1toT1" orientation="PtoT" inscription="a"
10       transend="//@pages.0/@nodes.0"
11       placeend="//@pages.0/@nodes.2"/>
```

Figure 4.13: A part of the XMI code representation of the CPN model presented in Figure 4.12.

Lines 5-7 describe the place `P1`, which is a node of type `Place` with the name `P1`, and the corresponding list of arcs that are connected with it. Lines 9-11 constitute the first occurrence of the tag `arcs` and describe the arc `fromP1toT1` flowing from place `P1` to transition `T1`, thus its has the orientation `PtoT` with the inscription `a`. This example illustrates how the target model in our transformation is built.

4.3.2 Details of the transformation

The mapping from scenarios described as sequence diagrams to a CPN model is implemented using ATL. Transformations in ATL are defined by helper operations and a set of rules, called transformation rules. An helper operation is defined in the context of a metamodel element, and it can be used within the rules. Each transformation rule matches model elements in a source model and creates elements in a target model.

This subsection presents an illustrative subset of the ATL rules to be applied to the elements in the source metamodel (sequence diagrams) to obtain the corresponding elements in the target metamodel (CPN modelling language).

Figure 4.14 presents an overview of the transformation, where the rectangles represent the artefacts being considered. The connections between the artefacts are either the relation "*conforms to*", represented by a continuous line with an arrowhead, or the relation "*is transformed into*", represented by a dashed line with an arrowhead. The model with the sequence diagrams is serialized in an XMI-file, which conforms to the metamodel for UML2 di-

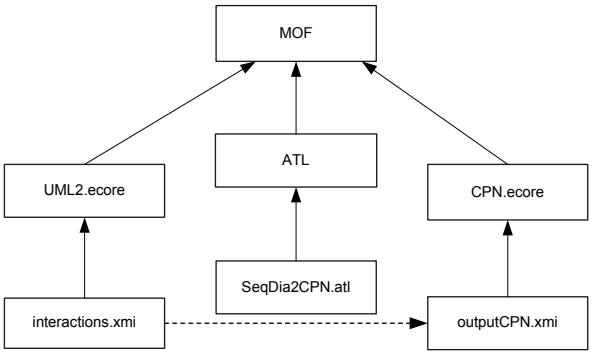


Figure 4.14: An overview of the transformation.

agrams described in ECORE. This model is used as a source model for the tranformation defined in the ATL file `SeqDia2CPN.atl`, which constitutes a model that conforms to the metamodel for the ATL transformations. As target model, one obtains a CPN model, which conforms to the metamodel for the CPN modelling language defined in Subsection 4.3.1. The metamodels can be seen as models conforming to MOF.

The rules to transform elements of sequence diagrams follow the ideas explained in previous sections, and each rule is defined for one of the elements of the metamodel for UML 2.0 sequence diagrams, whose details are described in Section 2.2.

Each message in an interaction has two message occurrence events: one corresponding to the sending of the message, and another corresponding to the receipt of the message. We consider that each message occurrence event in an interaction is transformed into a place in the CPN model. Figure 4.15 shows the part of the ATL rule to be applied to the message occurrences events in order to create the corresponding place. For this place, the colour set is defined as the name of the actor where the event occurs.

```
1 rule MOS {
2  from sourceMOS: UML!MessageOccurrenceSpecification
3  to targetPlace: CPN!Place (
4     name ← 'mos_' + s.name,
5     colorset ← 'cs_' + sourceMOS.covered.name )}
```

Figure 4.15: The ATL transformation rule MOS to be applied to a message occurrence specification.

A message of an interaction is transformed into a transition in the CPN model. Figure 4.16 shows the part of the ATL rule that permits to transform a message into a transition and the two arcs related with this transition. The name of the obtained transition is obtained by pre-appending the string "msg" to the name of the source message, and its guard is defined as true by default (see lines 9-10). The two arcs related with this transition are:

```
1  rule Msg {
2   from sourceMessage: UML!Message
3   to targetTrans: CPN!Transition (
4      name  ← 'msg_'+ sourceMessage.name,
5      guard ← 'true'),
6     arcToTrans: CPN!Arc(
7      name   ← 'arc_to_' + sourceMessage.name,
8      orientation ← #PtoT,
9      inscription ← 'var'+ sourceMessage.name,
10     transend    ← tragetTrans,
11     placeend    ← sourceMessage.sendEvent    ),
12    arcFromTrans: OUTMODEL!Arc(
13     name   ← 'arc_from_' + sourceMessage.name,
14     orientation ← #TtoP,
15     inscription ← 'var'+ sourceMessage.name
16     transend    ← tragetTrans,
17     placeend    ←
18      let nextMOS: ...
19      in if (nextMOS  →  oclIsUndefined())
20        then sourceMessage.receiveEvent.covered
21        else nextMOS
22        endif )}
```

Figure 4.16: The ATL transformation rule Msg to be applied to a message.

1. one input arc that comes from the place obtained by the transformation of the sending event associated with the considered message (see lines 6-11); and

2. one output arc that goes to the place obtained by the transformation of the message occurrence that follows the receiving event in the lifeline (see lines 12-22), where variable `nextMOS` receives the value of the next message occurrence specification in the lifeline. Then the value of `nextMOS` is tested in order to discover if there exists a next message occurrence event in the lifeline.

There exists a rule to map an interaction element into a CPN element, where the sets of nodes and arcs are created based on the transformation of the elements occurring inside the considered interaction.

A lifeline is represented by a place in the target model. Figure 4.17 presents the ATL code of the rule to be applied to a lifeline.

```
1 rule Lifeline {
2     from s : INMODEL!"uml::Lifeline" (
3       thisModule.inElements → includes(s))
4    to    t : OUTMODEL!Place(name ← s.name)}
```

Figure 4.17: The ATL transformation rule Lifeline to be applied to a lifeline.

The interaction is the top-most element of a sequence diagram, i.e., in the structure of a sequence diagram an interaction contains all the elements constituting the diagram. In this way, the rule for the transformation to be applied to an interaction outputs the corresponding CPN model, collecting the resulting CPN model elements (nodes and arcs) of the application of the corresponding rules to the elements in the sequence diagram. Figure 4.18 details the code to implement the rule of an interaction.

```
rule Interaction {
 from s: INMODEL!"uml::Interaction"
      (thisModule.inElements → includes(s))
 to t: OUTMODEL!CPN(
      name ← 'INTERACTION_'+ s.name ,
       nodes ← thisModule.allLifelines
           →  union(thisModule.allMessages)
           →  union(thisModule.
            allBehaviorExecutionSpecifications)
           →  union(thisModule.
            allMessageOccurrenceSpecifications),
      arcs ← (thisModule.allMessages
            →  collect(m|
             thisModule.resolveTemp(m,'arcToTrans')))
           →  union(thisModule.allMessages
           →  collect(m|
            thisModule.resolveTemp(m,'arcFromTrans')))
           →  union(thisModule.allMessages
           →  collect(m|
            thisModule.resolveTemp(m,'arc1'))
          ))
}
```

Figure 4.18: The ATL transformation rule Interaction to be applied to an interaction.

Chapter 5

Enriching CPN models for Animation

Summary

This chapter describes some guidelines that we suggest to be taken into account when enriching a CPN model for animation purposes. These guidelines aim to ensure that the obtained CPN model: (1) allows the easy modification of the initial conditions of scenarios; (2) includes a description of the state of the relevant entities in the environment; and (3) implies that the elements related to the animation are clearly separated from the other ones.

Contents

5.1 Introduction 78
5.2 Mapping sequence diagrams into a CPN model 78
5.3 Data representation for the environment 81
5.4 Animation of messages in the sequence diagrams 83
5.5 Initial conditions for scenario execution 84
5.6 Building an animation 86

5.1 Introduction

The CPN model that is constructed from a set of scenario descriptions is an executable model that can be used to drive a graphical animation layer showing elements and concepts from the problem domain. Additionally, the CPN model needs to include a mechanism to manage how animation events are handled.

The BRITNeY [Westergaard 2006; Westergaard and Lassen 2006] animation tool is used to connect the execution of the CPN model in the CPN Tools with the SceneBeans objects corresponding to the animation specified in XML-based file. We use the SceneBeans plug-in present in the BRITNeY suite to display and interact with a SceneBeans animation.

As stated before, we consider that the requirements process includes the creation of a set of use cases. The behaviour of each use case is detailed by a collection of scenario descriptions, which can be represented by sequence diagrams. In the UML 2.0, sequence diagrams have many high-level flow operators. The transformation from these sequence diagrams to a CPN model is based on the general principles described in Chapter 4, which associate each message in the sequence diagram with a transition in the CPN model and define some mechanisms in the CPN model to represent the high-level operators present in the sequence diagrams.

We describe the construction of a CPN model to execute the scenarios described by sequence diagrams and also the additional constructs that need to be introduced to allow the obtained CPN model to regulate the animation. The description of the techniques is uses some examples taken from the case study of an elevator controller system described in Chapter 6, in particular we consider the sequence diagram with the main scenario for the "Service Floor" use case presented in Figure 5.1. The material presented in this chapter was published in [Ribeiro and Fernandes 2007a].

5.2 Mapping sequence diagrams into a CPN model

We suggest that each sequence diagram is translated to a CPN model where there is a substitution transition for each message or high-level operator in the sequence diagram. The places between substitution transitions guarantee the order between the messages in the sequence diagram, and their colour set needs to include the necessary information to allow the parallel execution of many scenarios.

Figure 6.27 shows a CPN model that was obtained from the sequence dia-

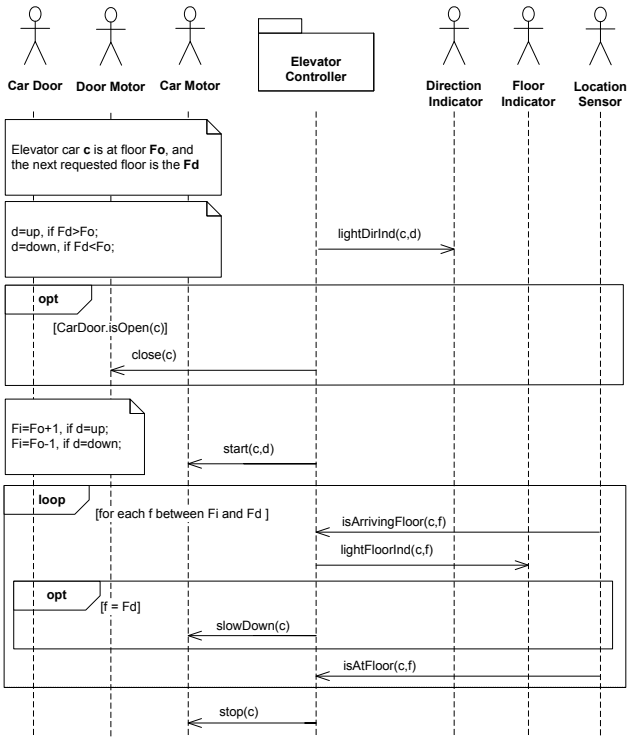


Figure 5.1: Sequence Diagram describing the "Service Floor" use case.

gram in Figure 5.1, where messages, and high-level operators in the sequence diagram are represented by substitution transitions in the CPN model.

For example, the message lightDirInd is represented by the substitution transition with the same name, and the first opt operator in the sequence diagram is represented by the substitution transition "opt (Is car door open?)". The messages inside the opt operator are tackled inside the corresponding subpage. The subpages contain the necessary details to animate the messages in the sequence diagram.

Places in the CPN of Figure 6.27 have the colour set ScenarioUC2, which provides the necessary information to execute a scenario of "Service Floor" use case, namely the origin and destination floors and the car being used. In other words, the definition of the colour set ScenarioUC2 comes from the textual annotation *"Elevator car c is at floor Fo, and the next requested floor is the Fd"*, from the sequence diagram in Figure 5.1, where the variables c, F_o and F_d are informally declared. These implicit and informal declarations of

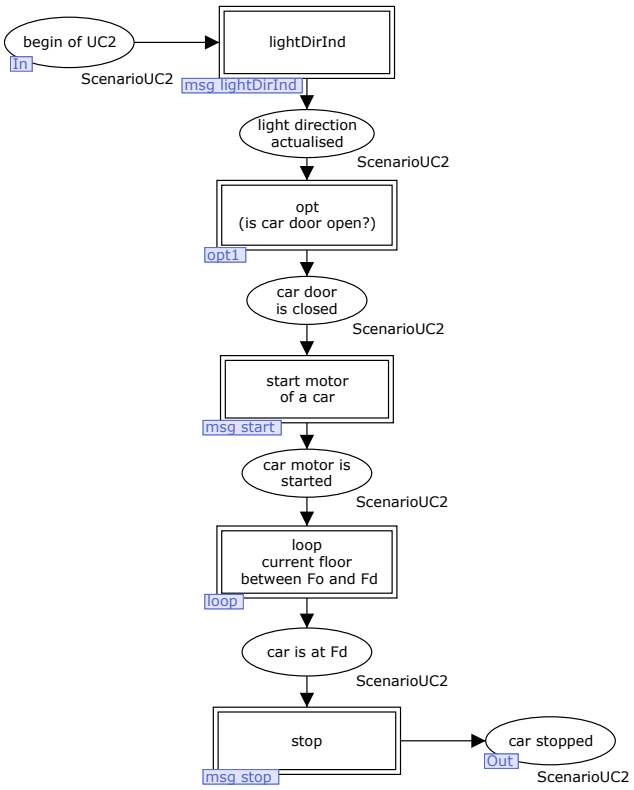


Figure 5.2: CPN model representing a sequence diagram for UC2 "Service Floor".

variables by a textual annotation allow for the usage of the same sequence diagram in different situations, by using the variables as a parameter in messages, or even in other textual annotations. The definition of the colour set ScenarioUC2 in the CPN ML programming language has the following code:

```
1 colset ScenarioUC2tmp = record
2                         c:  CarId *
3                         fo: FloorNumber *
4                         fd: FloorNumber ;

5 fun hasDiffFloors(s:ScenarioUC2tmp)= (#fo s) <> (#fd s) ;

6 colset ScenarioUC2 = subset ScenarioUC2tmp by hasDiffFloors;
```

The colour sets CarId and FloorNumber are defined as integers to identify a car and a floor, respectively. The colour set ScenarioUC2tmp is created essentially to be used in the definition of the colour set ScenarioUC2. When executing an instance of the "Service Floor" use case, it is implicit that the origin and destination floors are different. This is specified using the predicate hasDiffFloors to restrict the colour set ScenarioUC2tmp to obtain the colour set ScenarioUC2, whose elements are guaranteed to have different origin and destination floors.

The colour set ScenarioUC2 is used to distinguish between parallel executions of the use case, and thus the colour set must identify the car, the origin floor, and the destination floor. To start the execution of the CPN model it is necessary to define the input parameters using the place begin of UC2, where several tokens can be put to allow the parallel execution of different UC2 instances.

The places in the CPN model in Figure 6.27 ensure that the order between messages in the sequence diagram is maintained when firing the transitions in the CPN model. Each token in the place begin of UC2 means that a "Service Floor" has been requested to be executed, i.e., a given car must travel from a origin to a destination floor.

A token in the place light direction actualised means that the direction indicator light is now indicating the direction that the car is taking to go from the origin to the destination floor, and allows the next transition to be enabled.

5.3 Data representation for the environment

It is important to define a data representation of the main elements in the environment of the system under development. This data description is used to represent both the current state of the elements in the environment, and the changes introduced by the behaviour of each message in the sequence diagrams.

To obtain a description of the system's environment, its elements are specified as data in the CPN model using the CPN ML programming language, defining a colour set for each element in the environment. In our case study, the cars and the floors are the top-most entities of the environment. For example, the colour set Car is a record with an identification of the car, the door of the car, the motor to move the car, and the sensors and actuators inside the car.

```
1 colset Car = record
```

```
                id        : CarId       *
                motor     : CarMotor    *
                door      : CarDoor     *
                interior  : CarInterior ;
```

Where the CarId colour set is an integer, the CarMotor colour set is a tuple containing its moving speed and the direction being followed. The CarDoor colour set is defined as a record stating if the door is open or closed, and representing also the motor, the sensor and the timer of the door.

```
colset CarDoor = record
                     door      : Door      *
                     doorMotor : DoorMotor *
                     doorSensor: DoorSensor*
                     doorTimer : DoorTimer ;
```

Similarly, the entities inside a car are defined by the colour set CarInterior, which includes the lights to indicate the direction being followed by the car, the lights to indicate the current floor (there is a light for each floor), and a control panel where we can find a button to open the door, and buttons to allow the passenger to select one of the existing floors.

```
colset CarInterior = record
                       directionIndicator : Direction   *
                       floorIndicator     : FloorNumber *
                       controlPanel       : ControlPanel;
```

We consider that the components in the car are part of the colour set, such as the car door, the door motor, the door sensor, the door timer and the car motor.

The textual annotation "$d = up,\ if\ Fd > Fo;\ d = down,\ if\ Fd < Fo;$" in the sequence diagram of Figure 5.1 is represented by the function calcDirection which takes a ScenarioUC2 colour set and gives a direction based on the origin and destination floors, and is defined as follows:

```
colset Direction = with up | down | idle ;

fun calcDirection(a:ScenarioUC2) =
    case (Int.compare(#fo p,#fd p)) of
       LESS    => up
      |GREATER => down
      |EQUAL   => idle ;
```

Some of the messages in the sequence diagram have the parameter direction, which can be calculated using the values of the ScenarioUC2. For example, the message lightDirInd uses the parameter direction.

5.4 Animation of messages in the sequence diagrams

One must detail how the execution of each substitute transition in the CPN model representing a message in the sequence diagram is animated in the SceneBeans animation.

The communication between the CPN model and the SceneBeans animation is done using an animation object which can be used in the code segments of CPN model to invoke some commands to be executed in the SceneBeans animation. The CPN model contains the following declaration of the variable `anim` as a SceneBeans object:

```
1 structure anim = SceneBeans(
2                  val name= "Elevator Controller Animation");
```

Figure 5.3 shows the subpage associated to the message lightDirInd. The animation of a message in the sequence diagram is divided into two transitions of the CPN model: the first one (lightDirInd) to invoke the command in the animation, and the second one (ack lightDirInd) to wait for the feedback from the animation, informing that the command has been executed. Thus, a token in the place waiting for lightDirInd event means that the animation is updating the appearance of the direction indicator in the animation. This mechanism, that waits for the event from the animation, ensures the synchronization between the animation and the execution of the CPN model.

To ask for an execution of a command in the SceneBeans animation, we use the "invokeCommand" method, which can be used for an object with a SceneBeans animation. The parameter for this method is a string that must correspond to a command in the animation. The command lightDirInd is used to animate the direction indicator in an elevator car has two parameters:

- the name of the car, and
- the direction that the car is taking.

To ease the creation of the command identifier, the following function can be created:

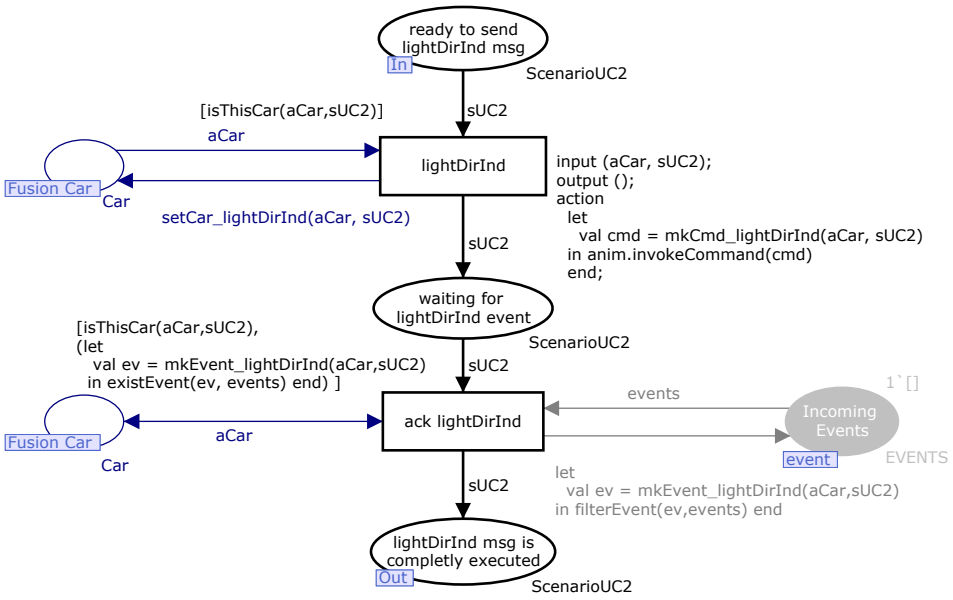


Figure 5.3: CPN model for the execution of the message lightDirInd.

```
1 fun mkCmd_lightDirInd(aCar:Car, sUC2: ScenarioUC2) =
2 let
3     val sc      = carName(aCar)
4     val  strd = showDir(calcDirection(#floors sUC2))
5 in "lightDirInd("^sc ^"Car,"^ strd ^")" end
```

To verify the incoming of the corresponding event of the command lightDirInd, it is necessary to test if the event is in the list of incoming events. Our CPN model is constantly aware of the events being generated by the animation through the module in Figure 5.4. The running fusion place is used to test if the animation has already started.

5.5 Initial conditions for scenario execution

We suggest to add a module to the CPN model where the initial conditions for the scenario execution and the selection of the SceneBeans XML file to used are introduced.

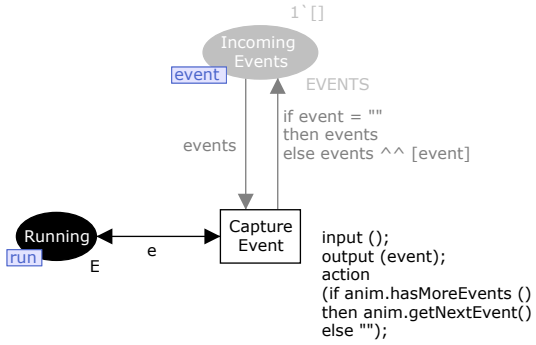


Figure 5.4: Subpage of CPN model to capture events from the SceneBeans animation.

The CPN model in Figure 5.5 analyses the initial conditions of the environment that the user wants to simulate and subsequently initialises the animation. The initial conditions are introduced in the pre-places (in green) of transition Initialise. The firing of the Initialise transition invokes the necessary commands to start running the animation according to the specified conditions. Figure 5.5 constitutes a top page of the CPN model, where for each scenario a separate subpage exists and for which the start place can be found through a fusion place. The Running place is used to allow the execution of the CPN module in Figure 5.4.

Let us see now how to change the CPN model to allow the consideration of a different numbers of cars and floors in the animation. To accomplish this, we need to adapt both the CPN model and the SceneBeans XML file defining the animation layer.

The changes in the CPN model are done in the topmost page presented in Figure 5.5, changing the initial marking of the pre-places for the Initialise transition. It is also necessary to change the value of a constant that represents the number of floors and another that represents the number of cars. The post-places for the Initialise transition are fusion places connected to CPN modules representing the behaviour for scenarios. These changes must be complemented with the corresponding changes in the animation specification, namely in what concerns the variables that represent the available cars and the available floors.

Assuming that we want to introduce a new car in the middle of the two existing cars, we define a constant, e.g. centerCar, with its data representation, to be included in the list of initial marking in place Cars of Figure 5.5,

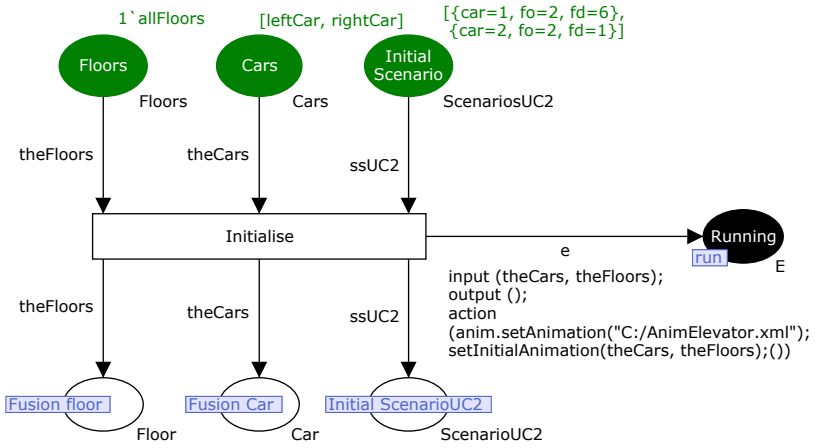


Figure 5.5: CPN module to initialise the environment values and the SceneBeans animation.

which results in the list [leftCar, centerCar, rightCar]. If we want to execute a scenario related to this new car we need to add the scenario description to the list of initial scenarios to be executed, and to introduce in the animation the icon for the additional car.

5.6 Building an animation

This section describes the tools used to deploy animations, and how to use them.

5.6.1 Initial considerations

The visual animation to be built is intended to reproduce the behaviour of the elevator controller, specified by the considered collection of sequence diagrams. The animation can be used during the validation to facilitate the dialogue between the system developers and the users and clients, which is a useful mechanism to ensure that both parts agree on what is to be developed.

One can start the construction of a visual animation, after some initial work on the analysis task has been accomplished. For this purpose, it is useful to have the context diagram, since it enumerates the main entities of the environment with which the controller interacts. Typically, these entities

are strong candidates to be represented in the animation, since the system's behaviour depends on them.

It is also important to have the use case descriptions, together with their corresponding sequence diagrams, since they describe which messages are received by the elements in the environment, and how these elements react on those messages. These artefacts specify the behaviour that must be covered by the animation layer.

The construction of the animation must be synchronised with the activity of creating the CPN model, because the CPN model must include some elements which are specific to control the animation as explained in Subsection 5.6.2.

The animation was created using the SceneBeans tool [Magee et al. 2000; Pryce and Magee 2007], which is a framework for creating and controlling animations, using the Java programming language. There is a XML-based file format to define animations and a parser to translate those XML files into animation objects for SceneBeans.

In the SceneBeans architecture, scene graphs, behaviours and animations are the basic elements with which one can create a visual animation. A *scene graph* is implemented in JavaBeans by a direct acyclic graph, which draws a two-dimensional image. In the leaf nodes of a scene graph there are primitive shapes (such as circles, ellipses, rectangles). An intermediate node either combines or transforms its subgraphs. The combination can be done in two ways: putting one subgraph on the top of another; or choosing one from the set of subgraphs. In a transform node, it is possible to apply a linear transformation followed by a translation to its subgraphs (for example rotation, scaling or translation) or to change the way that its subgraphs are rendered (for example, changing the colour in which a node is drawn).

Associated to each graphic element in the animation, there are some behaviours (not to be confused with the behaviour of the controller) to animate some of the properties of the element. A *behaviour* in SceneBeans is implemented by a Java bean that controls a time value, and when the value changes it announces an event. This permits the animation of the visual appearance of the scene graph. Notice that there is a so called *animation* thread that manages the frame rate of the overall animation and signals the passage of time. One can also to define commands to call the execution of a set of behaviours and to announce an event when they finish.

5.6.2 Static part of the animation

The behaviour of reactive systems, in general, lies in the interaction with its environment, by sending messages to the environment, which in response can also send messages in the opposite direction (i.e., to the controller). The elements in the environment can be seen as the actors of the system being developed. Some of these elements must be depicted in the animation. The first activity that must be done is to select a picture to represent each element in the environment.

The validation focuses essentially on the reactions of the controller to requests made by the passengers. If some flaw on the behaviour of the elevator is detected during validation, the developer may also need to analyse all the entities of the environment (even those that do not appear in the visual animation), to identify and understand the cause of the error.

The structure of the animation is essentially based on importing some icons to represent an element of the system or on drawing a geometric artefact using a XML tag. This structure is present in the leaf nodes of the scene graph, and for example to include the image in the file door.png we use the XML tag primitive, as follows:

```
1 <primitive type="sprite">
2     <param name="src" value="door.png"/>
3 </primitive>
```

The primitives allow the definition of the elements to be used in the dynamic part of the animation.

5.6.3 Dynamic part of the animation

In this subsection we show how to define the dynamic part of an animation in the SceneBeans XML-based format.

As we said before, in the leaf nodes of a scene graph there are primitive shapes and in the intermediate nodes either combination or transformation of its subgraphs using a set of parameters. Each parameter can be associated to a behaviour that needs to be previously defined. To allow the user to control the existing behaviours, there are commands, and each one includes a sequence of behaviour invocations. Thus, calling a command results on the animation of some of the parameters present in the intermediate nodes.

This message allows the Direction Indicator to change its state, among its possible values (up, down, and idle). The Direction Indicator is composed of two triangular lights. If the car is going up (down), the top light "$\bigtriangleup$"

is on (off) and the bottom light "▽" is off (on). If the car is stopped, the idle direction is represented by delighting both lights. Figure 5.6 presents three different images used to represent the possible states of the direction indicator.

(a) (b) (c)

Figure 5.6: Three representations for the light direction indicator inside the elevator car. (a) Going Up, (b) Going down, (c) Idle.

In the animation, it is necessary to define behaviours associated to the elements in the animation. For example, to indicate that the car is going up, we use the following XML code, whose XML tags param in lines 2-4 are used to set the parameters from, to and duration of the behaviour:

```
1 <behaviour algorithm="move" event="ldi"
2         id="showlightDirInd(rightCar,up)">
3    <param name="from" value="-1000"/>
4    <param name="to" value="750"/>
5    <param name="duration" value="0.0001"/>
6 </behaviour>
```

This block of code defines a behaviour that is based on a specific movement of an animation icon from a given point ('from') to another point ('to') during a given time ('duration'). This behaviour will be associated to the parameters in the nodes of the scene graph.

To indicate, during the animation, that the car is going up, one needs to put an icon (showing the top light on, and the bottom light off) at the position of the Direction Indicator entity in the animation picture. This is achieved by the following XML code:

```
1 <transform type="translate">
2     <param name="translation" value="(-1000,50)"/>
3     <animate param="x"
4         behaviour="showlightDirInd(rightCar,up)"/>
5   <animate param="x"
6       behaviour="hidelightDirInd(rightCar,up)"/>
```

```
7   <primitive type="sprite">
8       <param name="src"
9            value="direction_indicator_up.png"/>
10  </primitive>
11 </transform>
```

In lines 3 to 6, behaviours showLightDirInd (presented in the previous block of code) and hideLightDirInd are associated to the parameter "x" of the transformation node, in order to change the x-axis position of the icon, i.e., they move the icon horizontally in the animation picture. These behaviours are invoked through their inclusion in a command definition as we present in the next block of code.

There is one icon to represent each state of the Direction Indicator entity, and showLightDirInd behaviour moves the respective icon to a visible part of the animation, and the hideLightDirInd behaviour moves the respective icon to a non-visible part of the animation. Thus, the change to a new state is animated showing the icon of the new state and hiding the other two icons.

To allow the external invocation of these behaviours in order to animate the changing of the Direction Indicator in the car on the right-hand side to indicate the up direction, the following command announces an event with the same name.

```
1 <command name="lightDirInd(rightCar,up)">
2     <start behaviour="hidelightDirInd(rightCar,down)"/>
3     <start behaviour="hidelightDirInd(rightCar,idle)"/>
4     <start behaviour="showlightDirInd(rightCar,up)"/>
5     <announce event="lightDirInd(rightCar,up)"/>
6 </command>
```

There are similar code blocks for the other two possible directions and for the other car.

5.6.4 Scripting language

We have created a Ruby script to facilitate the manipulation of the XML tags in the XML-file that specify the animation. Ruby [Ruby 2007; Thomas et al. 2004] is a scripting language that follows the principles of object-oriented programming. The Ruby script uses components from the library REXML [REXML 2007].

With Ruby, we can easily manipulate the XML to be generated, namely when repetitive parts of the XML code follow a given pattern. To generate

the XML code for the animation, a Ruby script was created in order to save some functions to be used on the generation of XML for animations, independently from the case being considered. For example, the code in line 26 creates an object of the class Behaviour which has the following definition:

```
1  class Behaviour
2    def initialize(id,algorithm,params,event,announce="no")
3      @id = id
4      @algorithm = algorithm
5      @params = params  # class BehaviourParams
6      @event = event
7      @toBeAnnounced = announce
8    end
9    def toXML(xb)
10     xb.behaviour("id" =>@id, "algorithm"=> @algorithm ,
11                  "event" => @event){
12       @params.toXML(xb)
13       }
14   end
15 end
```

This a simple definition of a Ruby class, where we can find the same parameters as the behaviour XML tag, and the definition of the method toXML to generate the corresponding XML code. For example for the behaviour described above we write:

```
1 Behaviour.new("showlightDirInd(rightCar,up)",
2     "move", BehaviourParams.new(-1000,750), "ldi")
```

The advantage is that we can use this class to have a less verbose (i.e., easier to read by humans) code to specify a behaviour. Consequently, if we repeat this process for all other XML tags present in the SceneBeans XML-based format, we obtain a less verbose way to define an animation layer.

We believe that this Ruby script constitutes an abstract way to deal, with the XML-based stuff, in particular it is an easy way to work with parameterisation in the visualizations. When comparing it with the forall construct, we believe that the Ruby script is a more flexible and user-friendly way to manipulate the parameters. However, the forall construct has the advantage that it is defined in the same XML-file as the rest of the animation definitions.

Chapter 6

Case Studies

Summary

This chapter presents the case studies that were considered in this thesis: (a) the reactor system; (b) the elevator controller system; and (c) the check-in system in an international airport. These are different examples of reactive systems. The reactor system does not interact with human actors. In the elevator system there are human actors that interact with the system through the interface elements. Human actors are an important part of check-in system, thus it is crucial to consider their behaviour. The rules and the guidelines introduced in previous chapters are discussed and are exemplified in the context to the three case studies.

Contents

6.1 Reactor System 94
6.2 Elevator Controller System 125
6.3 Check-in System 141
6.4 Discussion 153

6.1 Reactor System

This section introduces the reactor system case study, which consists in a reactor that controls the filling of a tank. This case study was already considered in previous works [Adamski 1987; Fernandes et al. 1995; Machado et al. 1997].

The section illustrates two different ways to obtain CPN models for describing the reactor system. The first CPN model is obtained directly from the requirements (or more precisely adapted from an existent PN-based specification) and the another CPN model is obtained from sequence diagrams using the transformation rules described above in Chapter 4.

6.1.1 General description

The reactor is used to mix two liquids (or products) in specific quantities and to transport the mixed solution to an unloading area. A plant of the reactor system is presented in Figure 6.1. The system has two storage vessels, called

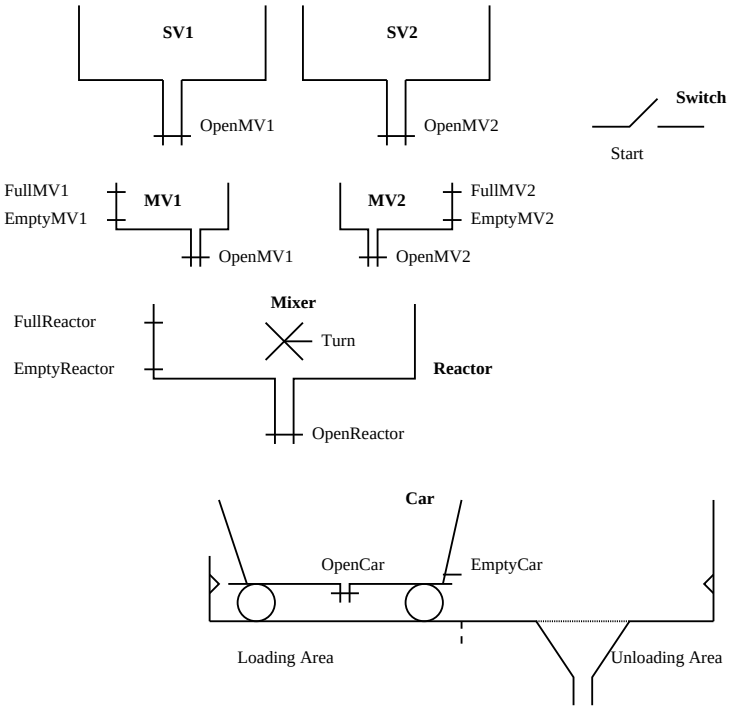


Figure 6.1: The environment of industrial reactor system.

SV1 and SV2, each one equipped with a valve (OpenSV1 and OpenSV2) to control the exit of the liquid. Below each storage vessel, there is a measuring vessel (MV1 and MV2), which has the same structure as the storage vessel, plus two sensors, one indicating when it is full (FullMV1 and FullMV2 sensors) and another one indicating when it is empty (EmptyMV1 and EmptyMV2).

The Reactor constitutes the main tank of the system, and it is fed with two kinds of liquids from the measuring vessels MV1 and MV2. To ensure complete reaction the liquid in the reactor is agitated by stirrer Mixer. After the reaction between the liquids is complete, the liquid in the reactor is discharged into the carriage Car. When the Reactor is empty the mixed product is transported using the Car.

When the button Start is pressed, the valves OpenSV1 and OpenSV2 are opened and measuring vessels MV1 and MV2 are refilled until sensor FullMV1 (FullMV2) is activated. Afterwards, OpenSV1 (OpenSV2) is closed. To start mixing, liquids are delivered into the reactor from the measuring vessels MV1 and MV2, by opening valves OpenMV1 and OpenMV2. Meanwhile, the reactor stirrer may start (Turn), when the level in the reactor is higher than MixingLevelReactor. When EmptyMV1 and EmptyMV2 get activated, OpenMV1 and OpenMV2 are closed and the reactor is emptied by opening OpenReactor. After discharging the reactor (which is given when EmptyReactor is activated), the product is transported by using the carriage Car, which is able to move right, to go the unloading area, and to move left, to return to the loading area.

6.1.2 A shobi-PN based CPN model

Based on the shobi-PN model of reactor system presented in [Machado et al. 1997], we have created the CPN model in Figure 6.2. The shobi-PN modelling language was conceived as an extension to SIP-nets (Synchronous and Interpreted Petri nets) [Fernandes et al. 1995]. The shobi-PN modelling language includes the same characteristics as the SIP-net modelling language, in what concerns synchronism and interpretation, and adds two new concepts by supporting object-oriented modelling mechanisms and a new hierarchical constructor, in both the control unit and the plant (i.e., the system under control).

The objects are represented by record colours, and the methods are represented by functions on the object's colour, e.g., the storage vessel object and the method to open it are defined by the CPN-ML code in Figure 6.3.

Notice that the token associated to each instance of an object is used to control the behavior of the CPN model. The place anti-place (a pre-place

of `t1`, and a post place of `t10`) is only used to restrict the firing of transition `t1`. To simulate the system behaviour, a CPN to represent the environment of reactor was created.

As previously described, the switch available in the reactor system can be pressed in order to start the system execution. The behaviour of the switch is modelled by the CPN module in Figure 6.4. The transition `Toggle Switch` represents the switch being toggled, and the place `Swt`, which is a fusion place with the place `Pf2` (that appears on the top of Figure 6.2), holds the current state (`on` or `off`) of the switch. This CPN module is complemented with the CPN-ML code in Figure 6.5. Line 1 defines the colour set `Switch` as an enumerated type with two alternative values (`on` or `off`). Line 2 declares the variable `aSwitch` with the colour set `Switch`. Line 3 defines the function `toggle` that toggles the value associated to the switch. Line 4 declares the predicate `isOn` to inspect if the switch is currently on.

After the switch `Start` is pressed the valves `openSV1` and `openSV2` are opened. At this point, based on the physical layout of the system, each measuring vessel starts being refilled with the liquid that comes from the corresponding storage vessel. Figure 6.6 presents a CPN module to represent the filling of measuring vessels. The activation of sensors `isEmptyMV1` and `isEmptyMV2` is represented by the transition `Begin Filling Measuring Vessel`, and the activation of sensors `isFullMV1` and `isFullMV2` is represented by the transition `End Filling Measuring Vessel`. The fusion places allows the connection of this module with the main module of the reactor system. During the simulation of the CPN model, the tokens in each place guarantees the correct order among the firing of these two transitions, and the time space between the firings of these two transitions represents the time necessary to filling the measuring vessel. The predicate `isConnected` (used as a guard of transitions in the CPN module) is defined based on the layout of the system to test the correspondence between a storage vessel and the measuring vessel that is below it (see Figure 6.1). The activation of `isFullMV1` (`isFullMV2`) implies that the valve `OpenSV1` (`OpenSV2`) must be closed. This is the reason why the transition `End Filling Measuring Vessels` takes a storage vessel `aSV` and puts back a modified one using the arc inscription "`StorageVessel.set isOpen aSV false`".

The opening of the valves `OpenMV1` and `OpenMV2` is represented by transition `t4` (in Figure 6.2) using the arc inscription `openAllMVs(listFullMVs)`. Afterwards, the measuring vessels can start to be emptied. Figure 6.7 presents a CPN module to capture the behaviour of emptying a measuring vessel, and delivering the liquids into the reactor. The transition `Begin Emptying Measuring Vessel` represents the start of emptying the measur-

ing vessel, deactivating the corresponding `isFull` sensor (`isFullMV1` or `isFullMV2`), and the transition `End Emptying Measuring Vessel` represents the activation of the corresponding `isEmpty` sensor (`isEmptyMV1` or `isEmptyMV2`), and consequently terminates the emptying of the measuring vessel.

Similarly, other behaviours of the environment are also modelled, such as the mixing of the liquids in reactor, the emptying of the reactor, the filling of the car, and all possible movements of the car.

The obtained CPN model has a similar behaviour when compared with the original shobi-PN for the reactor system. The objects in the shobi-PN are modeled as a colour set in the CPN model, and the methods over the object as functions on the colour set elements. To capture the synchronism present in the shobi-PN model, each set of transitions than can be simultaneously enabled to fire are represented by a unique transition into the CPN model. This exercise has been done to explore the capability of the CPN modeling language to model reactive systems that have many interactions with the environment. In this way, the CPN model includes also the behaviour that happens in the environment, which is an important part to capture in the model when developing an animation of the system (see Subsection 6.1.4). The author has also conducted a study over the reactor system to formally verify some of its desired properties (see Subsection 6.1.5).

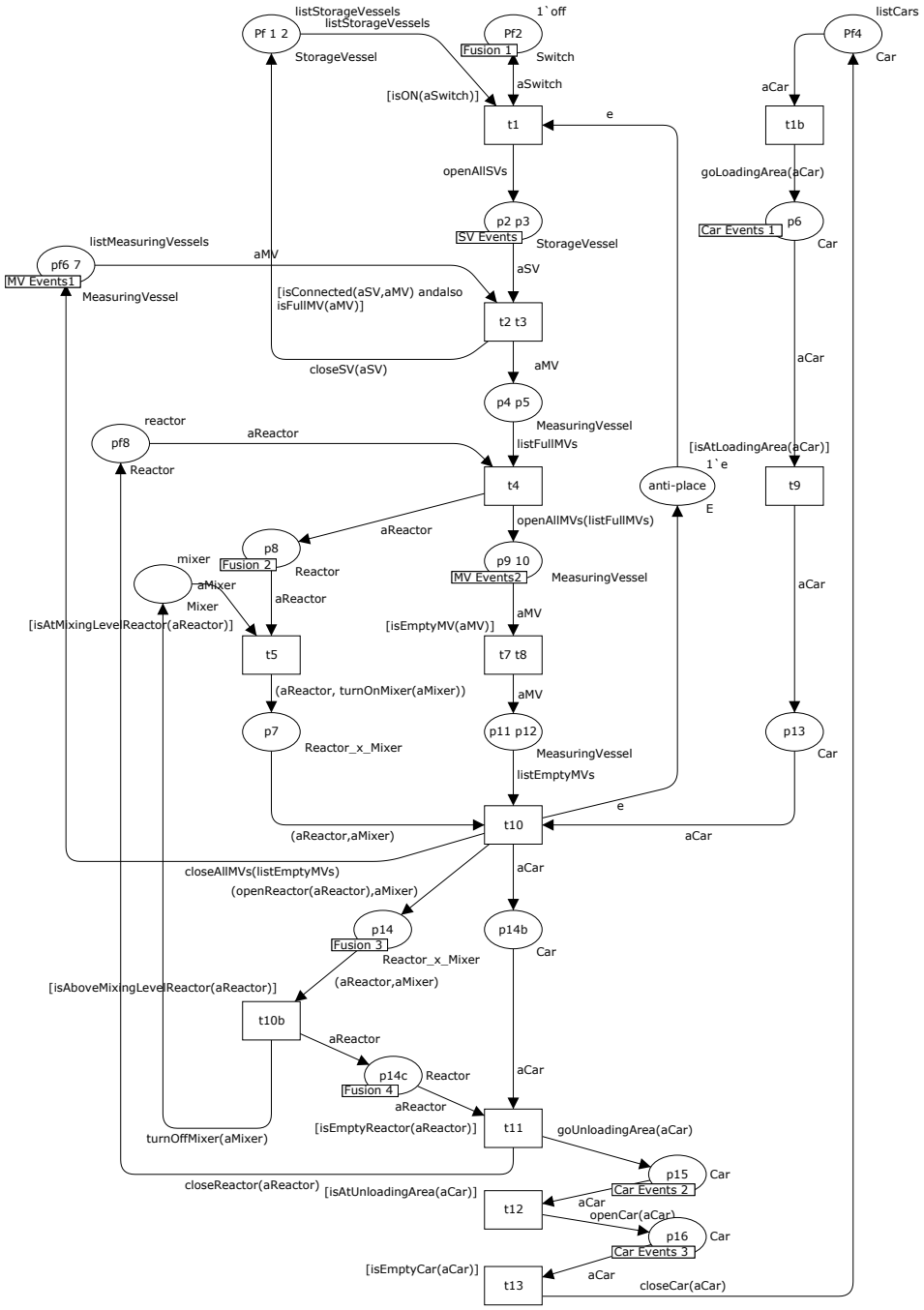


Figure 6.2: A CPN model for reactor system.

```
colset StorageVessel =
          record  id       : INT *
                  isOpen   : BOOL*
                  capacity : INT ;

fun openStorageVessel (sv:StorageVessel)
               = StorageVessel.set_isOpen sv true;
```

Figure 6.3: CPN-ML code for colour set `StorageVessel`.

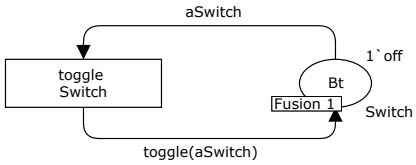


Figure 6.4: A CPN module for toggle switch.

```
colset Switch = with on| off;
var aSwitch : Switch;
fun isON(s:Switch) = if (s=on) then true else false;
fun toggle(s:Switch) = if (s=on) then off else on;
```

Figure 6.5: CPN-ML code for colour set `Switch`.

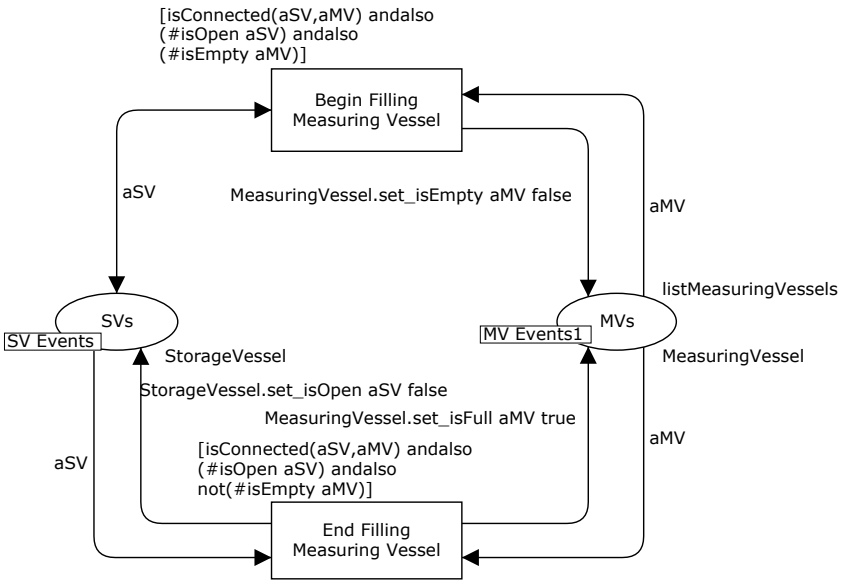


Figure 6.6: A CPN module for filling a measuring vessel.

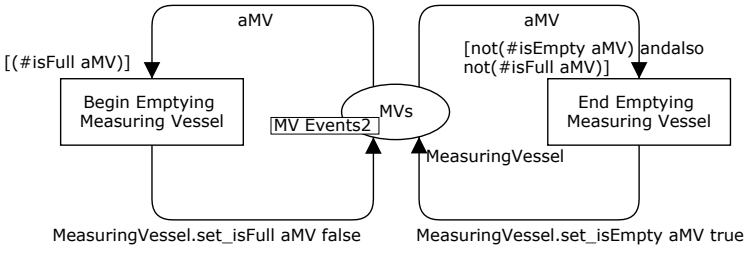


Figure 6.7: A CPN module for emptying measuring vessel.

6.1.3 A scenario-based CPN model

This subsection discusses how a CPN model for the reactor system can be obtained by applying the rules presented in Chapter 4 to a set of scenarios represented by UML sequence diagrams.

In this case, we assume that from the requirements document, one can construct sequence diagrams to represent some scenarios of the system's usage. An example is the sequence diagram in Figure 6.8 that uses high-level operators, namely the `ref` to point to another two sequence diagrams: "Preparing Car" (see Figure 6.9) and "Vessels Behaviour" (see Figure 6.10). High-level operators make it possible to represent in a unique sequence diagram several simple scenarios.

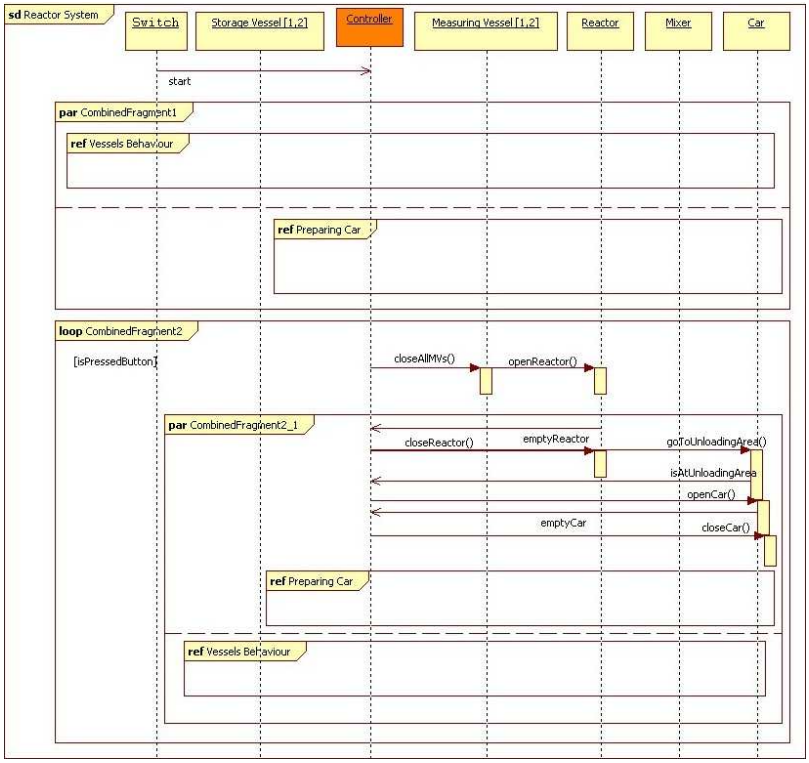


Figure 6.8: A sequence diagram describing some scenarios of using the reactor system

To transform this sequence diagram we firstly apply the rules to the fragments within a high-level operator. Afterwards, we compose the obtained

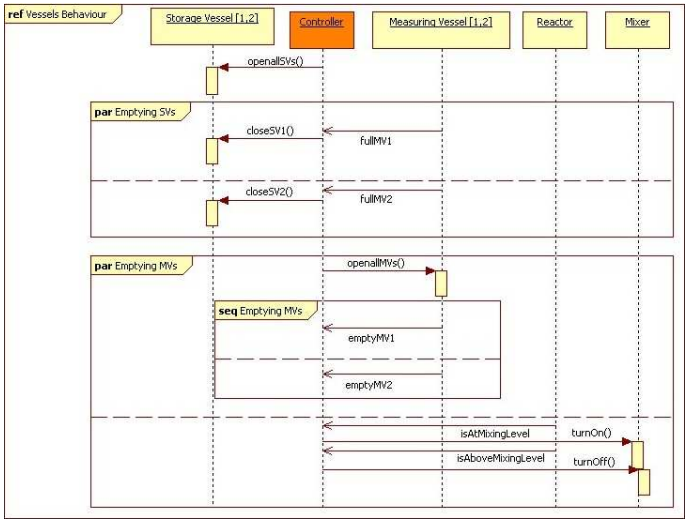


Figure 6.9: A sequence diagram describing the behaviour of vessels.

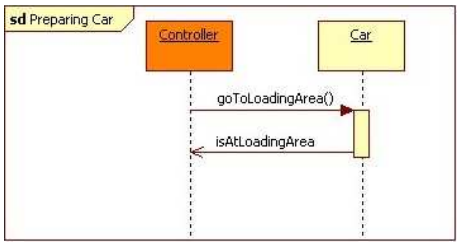


Figure 6.10: A sequence diagram describing the preparation of car.

CPNs into a hierarchical CPN. We put each sequence diagram pointed by `ref` into a subpage. Figure 6.11 shows the CPN obtained from the sequence diagram in Figure 6.9, where we can observe some transitions which are links to a CPN modules. The subpage `Vessels Behavior`, presented in Figure 6.11, corresponds to the sequence diagram presented in Figure 6.9.

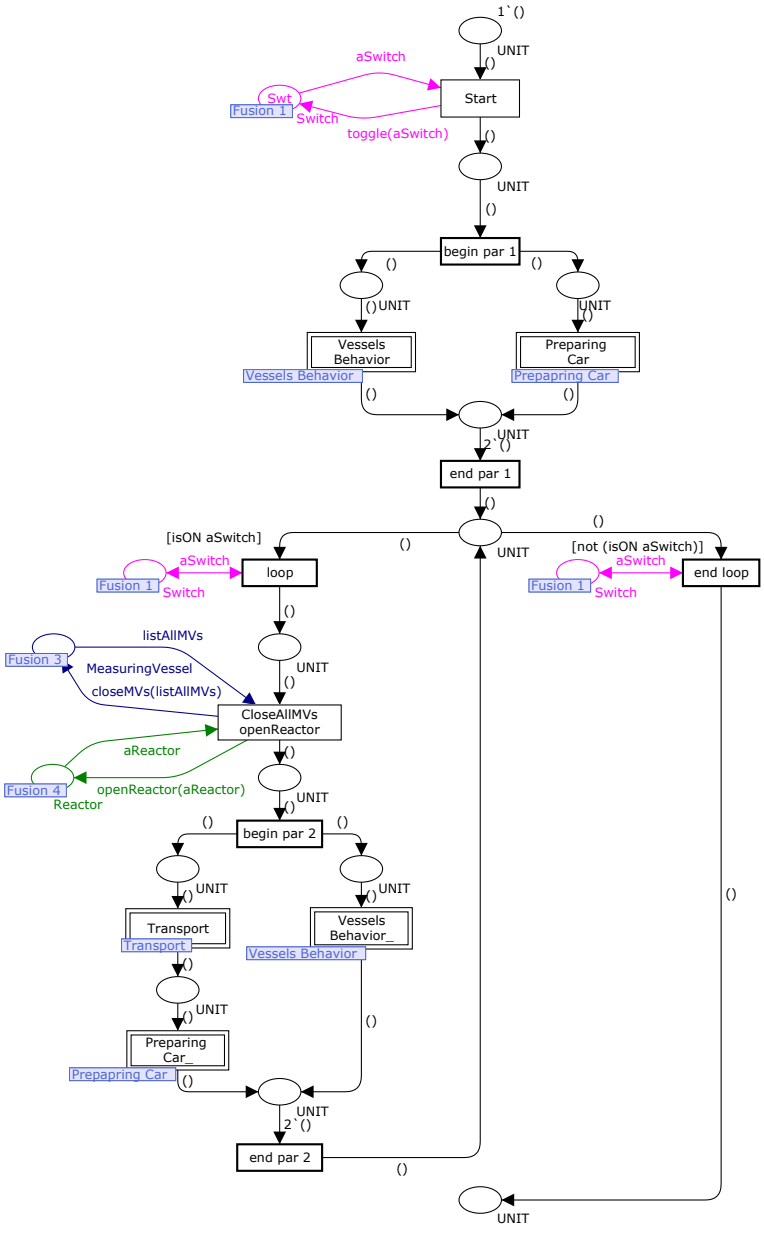


Figure 6.11: CPN from the sequence diagram presented in Figure 6.8.

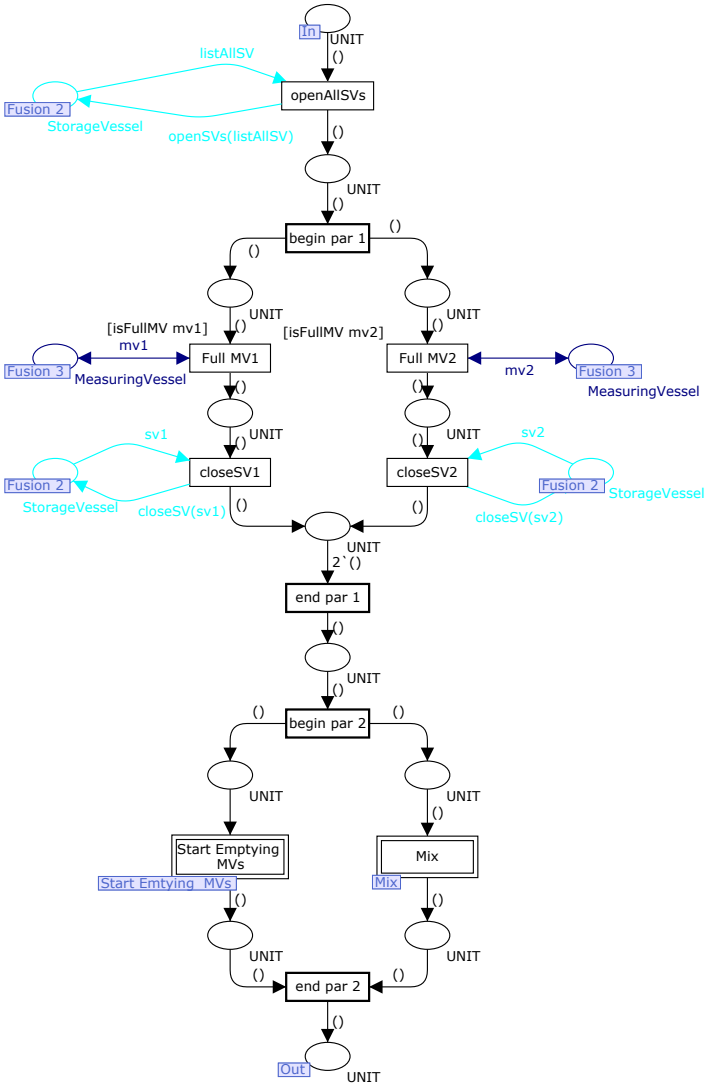


Figure 6.12: CPN to represent the behaviour of vessels (see Figure 6.9)

6.1.4 Building an animation

We have developed a SceneBeans animationto be associated with the created CPNs, through BRITNeY animation tool [Westergaard and Lassen 2005]. A screen shot of the animation of the reactor system is shown in Figure 6.13.

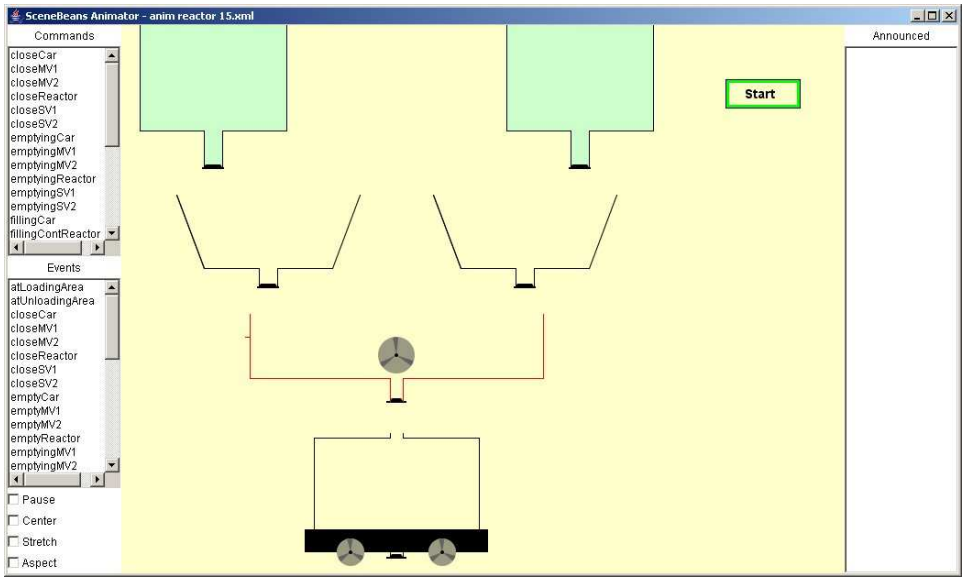


Figure 6.13: A screenshot of the animation of the reactor system

In this animation we can find the elements of the problem domain, which is the reactor domain. We have obtained this animation using the SceneBeans animator. On the top left side of the image we have commands accepted by the animation. On the bottom left side, we have the events produced by the animation. These sets of commands and events are used to do the interaction between the animation and the CPN models. For example, to animate the message "`openAllSVs`" in sequence diagram of Figure 6.9, we invoke, in the corresponding transition, two commands of the animation "`openSV1`" and "`openSV2`". The user can interact with the animation using the start switch. The vessels on the top are the storage vessels. Each storage vessel has a corresponding measuring vessel. In the center we have the reactor vessel, with a mixer inside of it. On the bottom, we have the car which transports the liquid to the unloading area.

This animation is intended to help the developers and users in validating

the behaviour of the reactor system.

6.1.5 A SIP-net approach to model the reactor

This subsection presents an exercise to model the reactor using a synchronous Petri net model (called SIP-nets) that was conducted before the experiments described in the previous subsections. This exercise was enlightening in the sense that it provide insights on the need to address the behaviour of the environment. The material presented in this subsection was published in [Ribeiro and Fernandes 2007c].

This subsection describes an approach to apply the model checking technique using the Spin tool. This approach allows important properties of reactive systems, such as liveness, deadlock-freedom, and the absence of structural conflicts among transitions, to be verified. The need to verify properties of a computer-based system is of paramount importance, namely when the systems are safety-critical.

In the proposed approach, the behaviour of the reactive systems (i.e., its control part) is modelled with a variant of Petri Nets, called SIP-net. When compared to traditional PN models, SIP-net models present guards associated to transitions, inhibitor and enabling arcs, and synchronous firings of the transitions. Due to this synchronous nature of the firings, the description of the SIP-net models with PROMELA is not trivial, because the Spin tool assumes the existence of a finite-state system where only one transition fires at each instant.

Since the values of the input signals are not known in the modelled system, we must consider all their possible values. In the SIP-net models the input signals are used as variables of guards. In this way, all possible scenarios for values of guards must be considered.

Therefore, this work discusses in some detail how SIP-net models should be specified with the PROMELA language (input format for the Spin model checker), so that some of their behaviour properties are verified. This model checking approach, namely the Haskell program, was already used in a case study [Ribeiro et al. 2005].

Synchronous and Interpreted Petri Nets (SIP-nets) were obtained by the enrichment of safe PNs with guarded transitions, synchronous firing, and also enabling and inhibitor arcs [Fernandes et al. 1997]. With these characteristics, the models that can be obtained are easier to synthesize [Adamski 1987]. In fact, synchronous circuits represent the largest portion of circuit designs and the state of the art in synthesising synchronous systems is more advanced and stable than the corresponding one for asynchronous

circuits [De Micheli 1994].

For developing embedded systems, the design process may benefit in some contexts from the usage of formal methods, namely to find critical errors and flaws, before final design and implementation decisions are taken. The Synchronous and Interpreted Petri Net (SIP-net) modelling language is considered in this approach to model embedded systems. This model of computation is based on safe Petri nets with guarded transitions and synchronous transitions firing, and also includes enabling and inhibitor arcs. The Spin tool, whose input language is PROMELA, is a verification system based on model checking techniques. This section presents a program to translate SIP-net models into PROMELA code and discusses in detail the adequacy of the created PROMELA specification for verification through model checking techniques.

Concurrency is considered one of the essential features of reactive systems [Manna and Pnueli 1992]. A concurrent system is a collection of sequential processes that in abstract are executed in parallel, i.e., it is not required that a separate physical processor is used to execute each process. A process is a set of instructions in a programming language which are executed sequentially. Thus, the semantics of concurrent systems is usually based on the notion of a global state, where the instructions of each process define a set of events denoting transitions between states. However, in a concurrent system an event affects and is affected by a limited number of other events (event scope). Events with disjoint scopes are free to occur independently.

When a formal model of a given system is created, it can be analysed with respect to some desired properties, thus allowing the detection of design errors prior to the system implementation. In general, the major weakness of PNs is the complexity problem; PN-based models tend to become too large for analysis even for a small real system [Murata 1989]. In [Fernandes et al. 1997] reachability graph analysis of the SIP-net description is used to investigate the properties of the modeled system.

Model checking [Clarke et al. 2000] is a verification technique that is based on the idea of exhaustively exploring the reachable state space of a system. The model checker Spin [Holzmann 2003] is a verification system, which accepts a specification language called PROMELA (Process Meta Language) [Holzmann 1991]. Spin has two main modes of operation: simulation and verification. Verification requires exhaustive search, whereas simulation does not and thus can deal with bigger state spaces. Simulation is a testing technique that can only indicate errors and never their absence. It is quite useful in practical terms, but in some situations verification is

the only solution, especially whenever one needs to formally guarantee that a system is free of errors, an essential condition for safety-critical systems. Spin uses the linear temporal logic (TL) to specify the properties to be verified.

The main motivation of this work is to study how SIP-net models can be verified using the Spin tool. We present a model checking approach, that uses the Spin tool to verify critical properties of embedded systems such as liveness, deadlock-freedom, and the absence of structural and behavioural conflicts among transitions.

The suggested design flow of our approach is presented in Figure 6.14. As tool support, we have created a computer application, written in the Haskell language, to automatically translate SIP-net models into equivalent PROMELA specification. With the assistance of the Spin tool, the generated PROMELA specification can be either simulated or verified with respect to some TL properties. This is possible, since the computer application can generate two different types of PROMELA specifications, one more adequate to be simulated and the other more adequate to the verification of some properties.

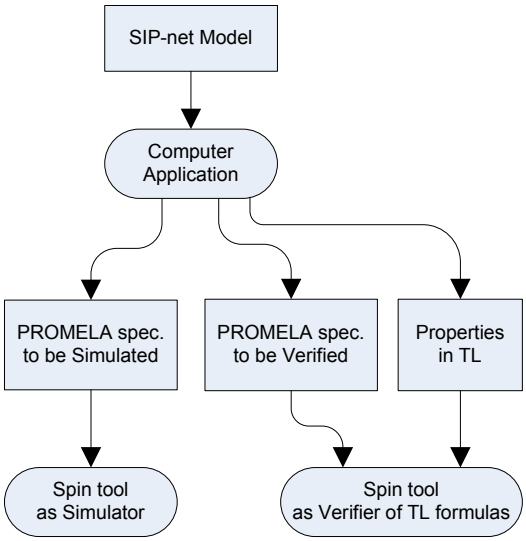


Figure 6.14: Design flow of the approach.

We already reported in [Ribeiro et al. 2005] the application of this approach in a case study. This work focus on the presentation of the issues

related to the translation process (from SIP-nets models to PROMELA specifications) and its tool support.

SIP-net Modelling Language

Next we introduce the SIP-net modelling language, showing how it was obtained from generic PN modelling languages. We present the structural apparatus to create a SIP-net model, the dynamics associated to an SIP-net model, and some important properties of SIP-net models. The definition of a generic PN, of type place/transition system, presented in [Reisig 1985] is the following.

Definition 6.1.1 (Petri Net) *A 6-tuple* $N = (P, T, F, C, M_0, W)$ *is a* ***Petri net*** *if and only if*

- (P, T, F) *is a* ***basic net****, i.e.* P*,* T *are disjoint finite sets and* $F \subseteq (P \times T) \cup (T \times P)$ *is a binary relation called* ***flow relation****, whose elements are called* ***arcs****. The elements of* P *and* T *are called* ***places*** *and* ***transitions****, respectively;*
- $C \colon P \to \mathbb{N} \cup \{\omega\}$ *gives the* ***capacity*** *(ω represents the infinite capacity) for each set;*
- $W : F \to \mathbb{N} \setminus \{0\}$ *gives one* ***weight*** *for each arc;*
- $M_0 \colon P \to \mathbb{N} \cup \{\omega\}$ *is the* ***initial marking*** *for net, respecting the capacities, i.e.,* $\forall_{p \in P}\ M(p) \leq C(p)$.

■

The SIP-net modelling language has been enriched with respect to the generic PN modelling language (see Definition 6.1.1) in three different ways:

- two new types of arcs are allowed: enabling arcs (also known as read arcs, test arcs, or positive context arcs), and inhibitor arcs (also designated negative context arcs) [Kleijn and Koutny 2000; Peterson 1981];
- transitions can have associated guards, which are propositional formulas where variables represent input signals of the modeled system, i.e., guards over transitions are formulas containing external variables, which may affect the enabling of transitions;
- transitions firing are synchronized with the active edge of a (global) clock.

Definition 6.1.2 (Structure of a SIP-net) *The **structure of a synchronous and interpreted Petri net** (written structure of an SIP-net model) is a tuple $N = (P, T, F, E, I, G)$ such that:*

1. *(P, T, F) is a **basic net**;*
2. *$E, I \subseteq P \times T$ are sets of enabling and inhibitor arcs respectively; the sets E, I and F are expected to be all disjoint;*
3. *$G : T \to PROP$ is a mapping associating a propositional formula to each transition.*

*Often P, T, F, E, I, G are denoted by P_N, T_N, F_N, E_N, I_N, G_N respectively. A **marking** to N is a mapping from P_N into the set $\{0, 1\}$.* ■

Graphically, the places and transitions are represented by circles and rectangles, respectively. The flow relation elements are represented by arrows.

The control part of embedded systems has input and output signals. Although the output signals are represented in the SIP-net model, associated with places, we do not consider them in the formalization of the SIP-net model. When analysing the behaviour of an SIP-net model, it is not necessary to consider the output signals, because we assume that their influence on the behaviour of the SIP-net model, if existent, is reflected by changes in the input signals.

Inhibitor arcs can be used to model a priority between processes. An inhibitor arc, represented by a dashed line with a circle, connects a place to a transition and disables the transition when the place is marked. Consider, for example, a system in which two processes access a shared resource. A conflict arises when both processes apply for the resource. To solve this problem, an inhibitor arc connecting the resource (a place) to one of the processes can be introduced.

Enabling arcs can be used to model a synchronization between two processes. An enabling arc, represented by a dashed line with an arrow, connects a place to a transition and enables the transition when the place is marked. Nevertheless, when the transition fires, no token is removed from the place connected to the transition through an enabling arc.

It is useful for each transition to determine the set of all places connected to it through a normal, inhibitor or enabled arc.

Definition 6.1.3 (Places linked to transitions) *Let N be a structure of an SIP-net model. For each $t \in T_N$:*

1. ${}^\bullet t = \{p \mid p \, \mathrm{F_N} \, t\}$ *is called the* ***preset*** *of* t*;*
2. $t^\bullet = \{p \mid t \, \mathrm{F_N} \, p\}$ *is called the* ***postset*** *of* t*;*
3. ${}^\triangleright t = \{p \mid p \, \mathrm{E_N} \, t\}$ *is the set of places linked with* t *through an enabling arc;*
4. ${}^\circ t = \{p \mid p \, \mathrm{I_N} \, t\}$ *is the set of places linked with* t *through an inhibitor arc.*

For $T' \subseteq T_N$*, let* ${}^\bullet T' = \bigcup_{t \in T'} {}^\bullet t$*,* $T'^\bullet = \bigcup_{t \in T'} t^\bullet$*,* ${}^\triangleright T' = \bigcup_{t \in T'} {}^\triangleright t$*, and* ${}^\circ T' = \bigcup_{t \in T'} {}^\circ t$*.* ■

As shown later, the behaviour of an SIP-net model is affected by the interpretation of the variables appearing in the guards of its transitions (external events). Thus, to simulate the behaviour of an SIP-net model, every possible valuation of the variables in the guards of its transitions must be considered. In rigour, we take a valuation of a net N to be a mapping from the set of all the variables occurring in the guards of N into the two-valued set $\{0, 1\}$. The set of all valuations to N is denoted by $\mathcal{V}_N$. In the next definitions we consider only conflict-free SIP-net models.

Now, we attend to the dynamics associated to an SIP-net model.

Definition 6.1.4 (Ready) *Let* N *be the structure of an SIP-net,* M *a marking to* N*,* $v \in \mathcal{V}_N$ *and* $t \in T_N$*. The transition* t *is* ***ready*** *for* M *and* v*,* $\texttt{ready}(\texttt{t}, \texttt{M}, v)$*, if*

1. *Each input place to* t *has one token, i.e.,* $\forall_{p \in {}^\bullet t} M(p) = 1$*;*
2. *Each output place to* t *have no tokens, i.e.,* $\forall_{p \in t^\bullet} M(p) = 0$*;*
3. *Each place connected to* t *with an enabling arc has one token, i.e.,* $\forall_{p \in {}^\triangleright t} M(p) = 1$*;*
4. *Each place connected to* t *with an inhibitor arc has no tokens, i.e.,* $\forall_{p \in {}^\circ t} M(p) = 0$*;*
5. *The interpretation of the guard associated with* t *using the valuation* v *is true.*

The set of all ready transitions for M *and* v *is denoted by* $T_{\texttt{ready(M,}v\texttt{)}}$*.*
The transition t *is* ***enabled*** *for* M*, written as* $\texttt{enabled}(\texttt{t}, \texttt{M})$*, when the first four previous conditions, which are related only with the places linked to the transitions, are verified. A set* $A \subseteq T_N$ *is said to be* ***ready****, denoted*

as $\texttt{ready}(A)$*, if there exists a marking* M *and a valuation* v*, such that all transitions in* A *are ready for* M *and* v*. The set* A *is* ***enabled****, written as* $\texttt{enabled}(A)$*, if there exists a marking* M*, such that all transitions in* A *are enabled.* ■

Although the condition 2) of definition 6.1.4 is not very often used, it is a way to guarantee the boundedness of net, because a net describing a control unit should be safe.

Usually, the firing in a classical PN is defined as the firing of one, and only one, ready transition at each time [Reisig 1985]. So, there is no simultaneous firings of transitions. The SIP-net token game differs in several ways from the standard one for PNs: instead of just one transition firing at a time, all transitions that are ready to fire must do so; firing of a transition is blocked if any place in its postset is non-empty. The use of simultaneous firing of sets of transitions (usually called steps) is discussed in [Mukund 1992].

Definition 6.1.5 (Simultaneous Firing) *Let* N *be the structure of an SIP-net model,* M *a marking to* N *and* $v \in \mathcal{V}_N$*. The marking* M' *to* N*, obtained from* M *with the valuation* v *through the* ***simultaneous firing*** *of all* $t \in T_N$*, ready for* M *and* v*, written* $\texttt{M}[\rangle_v\texttt{M}'$ *, is defined as:*

$$\forall_{p \in P_N} M'(p) = \begin{cases} 0 & if\ p \in {}^\bullet T_{\texttt{ready(M},v)} \\ 1 & if\ p \in T_{\texttt{ready(M},v)}{}^\bullet \\ M(p) & otherwise. \end{cases}$$

In other words, the input places of all enabled transitions become empty, one token is added to the output places of all enabled transitions, and the other places are left unchanged. ■

We can consider the above definition as a mathematical relation between two markings and one valuation. Thus the reflexive and transitive closure is defined as follow.

Definition 6.1.6 *Let* N *be the structure of an SIP-net, and* M *a marking to* N*. A marking* M' *to* N *is* ***accessible*** *from* M*, written* $\texttt{M}[*\rangle\texttt{M}'$*, if:*

1. $M = M'$*, or*
2. $\exists_{M''} \quad \texttt{M}[*\rangle\texttt{M}'' \ \wedge\ (\exists_{v \in \mathcal{V}_N} \quad \texttt{M}''[\rangle_v\texttt{M}')$.

The set of all markings to N *accessible from* M *is denoted by* $[\texttt{M}\rangle$*.*

*A **firing sequence** is a sequence with the form* $\mathtt{M_0}[\rangle_{v_0}\mathtt{M_1}[\rangle_{v_1}\ldots[\rangle_{v_{k-1}}\mathtt{M_k}[\rangle_{v_k}\ldots$ *where* $k \in \mathbb{N}$*, for all* i*,* M_i *is a marking to* N*, and* $v_{i-1} \in \mathcal{V}_N$*. By definition of simultaneous firing we can observe that a transition* $t \in T_N$ *can be fired in a firing sequence if there exists a natural* i*, such that* $\mathtt{ready}\,(\mathtt{t}, \mathtt{M_i}, v_i)$*.* ■

In order to have the behaviour of an SIP-net model completely defined it must be formed by a structure of an SIP-net plus a marking to this structure.

Definition 6.1.7 *A pair* $\mathbf{N} = (N, M_0)$ *where* M_0 *is the initial marking to* N*, is called an **SIP-net**.* ■

Some Properties of SIP-net models

Since transitions in SIP-net models fire synchronously, one may observe some conflicts amongst transitions, which are not present in other models of computation based on PNs.

Definition 6.1.8 *Let* $\mathbf{N} = (N, M_0)$ *be an SIP-net, and* $t_1, t_2 \in T$*. The transitions* t_1 *and* t_2 *are in **structural conflict** if transitions* t_1 *and* t_2 *have a common pre-place (post-place), i.e.,* ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ *(*$t_1{}^\bullet \cap t_2{}^\bullet \neq \emptyset$*).*

If the transitions t_1*,* t_2 *are in structural conflict, they are also in **behavioural conflict** if there exist an accessible marking* M *and a valuation* v *that enable both transitions.* ■

Let us now consider the formulation of the liveness property in the SIP-net modelling language. Often, the verification of liveness is very expensive and sometimes even impracticable. To overcome this difficulty, we follow the liveness levels proposed in [Murata 1989], some of which reduce the cost of verification.

Definition 6.1.9 *Let* $\mathbf{N} = (N, M_0)$ *be an SIP-net and* $t \in T_N$*. The transition* t *is **L0-live (or dead)** when it can never be fired for any marking accessible from* M_0*:*

$$\forall_{M' \in [\mathtt{M_0}\rangle} \quad \forall_{\mathtt{v} \in \mathcal{V}_R} \quad \neg\, \mathtt{ready}\,(\mathtt{t}, \mathtt{M'}, v);$$

The transition t *is **L1-live**, if there exists at least an accessible marking from* M_0*, for which* t *can be fired:*

$$\exists_{M' \in [\mathtt{M_0}\rangle} \quad \exists_{\mathtt{v} \in \mathcal{V}_R} \quad \mathtt{ready}\,(\mathtt{t}, \mathtt{M'}, v);$$

The transition t *is **L4-live (or live)**, if it is L1-live for all markings accessible from* M_0*:*

$$\forall_{M \in [\mathtt{M_0}\rangle} \quad \mathtt{L_1\text{-}Live}\,(\mathtt{N}, \mathtt{M}, \mathtt{t}).$$

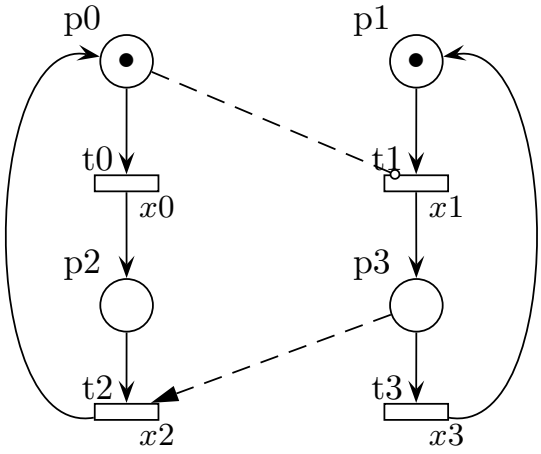


Figure 6.15: An example of an SIP-net model.

The SIP-net **N** *is said to be* ***Lk-live****, if every transition in the SIP-net* N *is* ***Lk-live****, where* $k = 0, 1, 4$. ■

We are not considering in this work the liveness levels L_2 and L_3, which means respectively for each transition t: given any natural m, t can be fired at least m times in some firing sequence; and t appears infinitely, often in some firing sequence.

An Example on Specifying SIP-net Models with PROMELA

In this section, we present a specification in the PROMELA language for the illustrative SIP-net model represented in Figure 6.15, which is based on the relation, through enabling and inhibitor arcs, between two similar SIP-nets.

Formally, this SIP-net model is defined by the tuple (N, M_0):

- $N = (P, T, F, E, I, G)$ is the structure of the SIP-net model, where: $P = \{p_0, p_1, p_2, p_3\}$, $T = \{t_0, t_1, t_2, t_3\}$, $F = \{(p_0, t_0), (p_1, t_1), (t_0, p_2), (t_1, p_3), (p_2, t_2), (p_3, t_3), (t_2, p_0), (t_3, p_1)\}$, $E = \{(p_3, t_2)\}$, $I = \{(p_0, t_1)\}$ and $G = \{(t_0, x0), (t_1, x1), (t_2, x2), (t_3, x3)\}$;

- $M_0 = \{(p_0, 1)(p_1, 1)(p_2, 0)(p_3, 0)\}$ is the initial marking.

The specification of this SIP-net model in PROMELA uses the following guidelines:

```
1  #define M 4
2  #define Nvar 4
3  #define rd_t0 (p[0] && !p[2] && v[0])
4  #define rd_t1 (p[1] && !p[3] && !p[0] && v[1])
5  #define rd_t2 (p[2] && !p[0] &&  p[3] && v[2])
6  #define rd_t3 (p[3] && !p[1] && !p[2] && v[2])
7  #define fire_t0 p[0] = 0; p[2] = 1;
8  #define fire_t1 p[1] = 0; p[3] = 1;
9  #define fire_t2 p[2] = 0; p[0] = 1;
10 #define fire_t3 p[3] = 0; p[1] = 1;
11 bool p[M];
12 bool v[Nvar];
```

Figure 6.16: Definitions and declarations in PROMELA.

- Each place is represented by a Boolean value and an array of Booleans, with length equal to the number of places, is used to represent the set of places;

- Each variable occurring in the guards is represented as a Boolean variable and the set of variables is represented by an array of Booleans with length equal to the number of variables.

With these guidelines, we present two approaches to obtain a PROMELA specification from an SIP-net model. These approaches differs on the way they treat with external variables occurring in the guards of SIP-net model. The first one uses an explicit representation of external variables. The second one does not use an explicit representation of variables, instead the semantics of variables are implicitly represented in the options of PROMELA specification, that is the second of the above guidelines will be not used.

Using Explicit Representation of the External Variables

The SIP-net model has four places ($p_0,\ p_1,\ p_2,\ p_3$) and four different variables in the guards ($x0,\ x1,\ x2,\ x3$). In PROMELA, we write the specification shown in lines 1, 2, 11 and 12 in Figure 6.16 to declare those places and guards. The array p represents the places ($p[i]$ is the place p_i of the SIP-net model) and the array v represents the set of variables ($v[j]$ is the guard xj of the SIP-net model) The definition of the enabling and the firing conditions for each transition uses arrays p and v. With these representations, ready conditions and firing rules for each transition are defined from lines 3 to 10 in Figure 6.16, where rd_ti represents the ready condition of transition ti and $fire\_ti$ represents the firing rule of transition ti.

According to the definition of simultaneous firing (Definition 6.1.5), all ready transitions at a given moment must be fired. Therefore, the

PROMELA specification must include one firing rule for each of the fifteen possible non-empty combinations of the three transitions.

There is one (and only one) choice in the do-loop for each subset of transitions. The choice condition for a subset T' of transitions is constructed by:

1. the conjunction of the ready conditions of all the transitions in T', and
2. the conjunction of the negation of ready conditions corresponding to each transition not in T'.

The do-loop may present non-determinism. In the above case we only have deterministic choices, because at most only one guard can be made true at a given moment. This means that we can not have two or more guards simultaneously true. Informally, given two guards, there is at least one ready condition of a transition t_i which appears uncomplemented in one of the guard and appears complemented (or negated) in the another one. Some of the guards are always false. We are not interested in considering the guards with all the ready conditions negated, because there is no action defined to that guard.

The behaviour of an SIP-net model depends on external stimuli through the variables occurring in the guards. In this PROMELA specification, the variables are present on the ready conditions (see lines 3 to 6 in Figure 6.16).

In an SIP-net model, nothing is known about the behaviour of its environment, more precisely the value of the variables are not modelled by the SIP-net model. Thus, we must consider all the possibilities for the value of each variable. In PROMELA, it can be done as illustrated in Figure 6.17, where the parameter ivar identifies the variable to be considered. The do-loop gives non-deterministically the value 0 or 1 to the considered variable.

```
1  proctype randomVar(int ivar)
2  {do
3     :: atomic{ v[ivar] = 0 ; }
4     :: atomic{ v[ivar] = 1 ; }
5   od }
6  init{ atomic{
7    run randomVar(0);run randomVar(1);
8    run randomVar(2); run randomVar(3);
9    p[0]=1; p[1]=1; p[2]=0; p[3]=0; /*Initial Marking*/
10   run procParSyn();}}
```

Figure 6.17: PROMELA process for specifying the SIP-net model example.

Running one process for each variable allows us to simulate the behaviour of the environment. The main process, for the SIP-net model example, firstly creates four instances of the randomVar process, one for each variable, (see lines 7 and 8 in Figure 6.17). After that, the places are initialized according to the initial marking. Finally, an instance of the procParSyn process is created. The corresponding PROMELA specification is presented in Figure 6.17.

To study the appropriateness of the PROMELA specification for the considered example, its behaviour has been simulated with the Spin tool. By running the interactive simulation, we have the possibility to choose one of two different values (0 or 1), for each of the guards $x0$, $x1$, $x2$, and $x3$. We have these options in all points of the simulation, which allow us to control the behaviour of the environment. In the PROMELA specification, both the controller and the environment run concurrently. Thus, we obtain a simulation of the system, whose non-determinism is introduced by external variables. In this PROMELA specification we are assuming that values of variables is part of system's behaviour, but this not the case for the SIP-net model, where the variables are considered elements of the environment.

Let us see what happens when we are using the Spin tool to analyse the obtained PROMELA specification. When simulating the PROMELA specification, since there is no explicit representation of environment's behaviour, it is possible to change the value of all variables in any point of execution, because as we said before the non-determinism is introduce by the value of each variable. In this way, when changing the variables we are defining the set of transitions to be fired. If none of the guards associated to the enabled transition, for the selected values, are evaluated to true we have a situations that the system we are simulating can not go to one its next states. When we are simulating this is useful, because we are only manipulating the environment and observing what happens to the system.

This solution is not adequate for verification of system's properties, because it is possible to obtain a path execution where there are only actions changing the value of one variable, which are not actions of the system we are considering. In this way when we try to verify a system property we have not the expected result. For example, when we try to verify on this PROMELA specification the liveness property of transition t_0, which is based on the inspection if for all states (in the system generated by the PROMELA specification) there is at least one next state validating the ready condition for transition t_0. When using the Spin tool to verify this property the obtained result is "not valid". Running the resulting "trail" simulation is based on an execution where the initial value of variable x_0 (false) is never changed,

```
1 #define en_t0 (p[0] && !p[2])
2 #define en_t1 (p[1] && !p[3] && !p[0])
3 #define en_t2 (p[2] && !p[0] &&  p[3])
4 #define en_t3 (p[3] && !p[1])
```

Figure 6.18: Definition of the enabled conditions.

and consequently the transition t_0 never becomes ready, because t_0 has the variable x_0 as its guard.

Not Using Explicit Representation of the External Variables

To overcome this problem, which prevents us from analysing the behaviour of the system, we adopt a different solution. The key idea of our approach to generate the PROMELA specification is that the semantics of guards are present, without explicitly representing the variables. This is possible if we guarantee that in each point of the PROMELA specification execution there are all the possible choices corresponding to the different valuations of the variables.

In the SIP-net model of Figure 6.15, transitions t_0 and t_1 are never enabled for a given marking, because there is a inhibitor arc from place p_0 to transition t_1. When analysing the SIP-model it is not necessary to consider that guards $x0$ and $x1$ are both true.

For a marking which puts a token into the places p_2 and p_3 (notice that this marking is accessible from the initial marking), the transitions t_2 and t_3 are both enabled. The variables $x2$ and $x3$ permit the sequential (non-concurrent) firing of transitions t_2 and t_3, i.e., transition t_2 can be fired without t_3 being fired at the same time (or vice-versa). This happens if $x2$ is true and $x3$ is false (or $x2$ is false and $x3$ is true). We still can fire these two transitions simultaneously if $x2$ and $x3$ are both true. We are not interested in the case that transitions $x2$ and $x3$ are both false.

Now, we can apply these ideas to the PROMELA specification for the considered example. Since in this solution there are no variables in the PROMELA specification, it is necessary to remove the references to the variables in the definition of the ready conditions. These new conditions only include references to places and are therefore called enabled conditions (see Definition 6.1.4). They have the PROMELA definition presented in Figure 6.18.

The structure of the main process is a do-loop, whose guards are based on the conjunction of the conditions corresponding to the enabled condition of a given subset of transitions.

v_i	$x0$	$x2$	$\neg x0 \wedge \neg x2$	Transitions
v_1	1	1	0	$\{t_0, t_1, t_2\}$
v_2	1	0	0	$\{t_0, t_1\}$
v_3	0	1	0	$\{t_2\}$
v_4	0	0	1	$\{t_3\}$

Figure 6.19: Truth values of guards.

In the solution that uses variables, when we have a valuation to the variables and a marking, we know that at most only one line of the do-loop has choice condition that holds true.

With one line in the PROMELA specification for each subset of transitions, we guarantee that the subsets of transitions, which are not expected to fire, have a false choice condition associated with them. In the solution without references to variables, we can not put one line for each subset of transitions. We select only the subsets of transitions, whose corresponding guards can be simultaneously true. For each $A \subseteq T_N$, we write $G_N(A)$ to denote $\bigcup_{t \in A}\{X : (t, X) \in G_N\}$. To write the PROMELA specification for the SIP-net model example, we select all the consistent sets in $\{G(A) | \emptyset \subset A \subseteq T_N\}$.

With respect to the SIP-net model in Figure 6.15, the sets of the variables in the guards are disjoint and guards could have the value true or false, so all the sets are consistent.

We most consider only the subsets of transitions which elements could be enabled for a given marking, otherwise we are generating a correct PROMELA specification but some of choices are "dead" code and so they could be removed.

We now change the guards of the transitions of the SIP-net model, such that: $G(t_1) = x0$, $G(t_3) = \neg x0 \wedge \neg x2$.

The variables occurring in the guards are $x0$ and $x2$. The variable $x0$ occurs in the guards of transitions t_0, t_1 and t_3, and the variable $x2$ occurs in the guards of transitions t_2 and t_3. In Figure 6.19, for each combination of the values for $x0$ and $x2$, the Boolean values of the guards are presented. The last column shows the set of transitions whose guards are validated by the valuation (v_i) in the row.

Although the guards of the transitions t_0, t_1 and t_2 are consistent, none of the transitions is simultaneously enabled with the other two transitions, due to the restrictions imposed by the structure of the SIP-net model. Thus, by the valuation v_1 in Figure 6.19, the transitions t_0, t_1, and t_2 can be fired alone. There exists a marking enabling transitions t_2 and t_3, but as we can

state in Figure 6.19 there is no valuation validating the ready condition for this two transitions. Thus the set of transitions $\{t_2, t_3\}$ is not considered in the do-loop of PROMELA specification.

Rules to Specify SIP-net Models with PROMELA

Based on the ideas expressed in the previous sections, we define more general rules in this section. A PROMELA specification is constructed from three basic types of objects: processes, data objects, and messages.

The principal process is called init. Many of PROMELA notational conventions derive from the C language, including declaration and initialization of variables. The do-loop statement gives a cyclic non-deterministic choice of one guard, and each guard has actions associated with it.

For a given SIP-net model, we define that the corresponding PROMELA specification has three parts: (1) the definition of the enabled condition for each transition, (2) the definition of the firing condition for each transition, and (3) the do-loop in the init function.

The first two parts are straightforward to obtain, because they only depend upon the structure of the SIP-net model. The do-loop is harder to obtain, because it must include all the possible subsets of transitions that may fire simultaneously. To define the alternative choices of the do-loop, some calculations must be performed based on the guards of each transition.

Firstly, we calculate the subsets of transitions which may be simultaneously enabled. Notice that subsets of this set are still sets of enabled transitions, and thus, we need only to consider the maximal subsets of enabled transitions, independently of the valuation of their guards. Let us denote the set of such maximal subsets by $PEMAXS$.

Secondly, for each $MT \in PEMAXS$, we calculate its maximal subsets of ready transitions, i.e., we consider the maximal subsets of MT whose guards can be made true under the same valuation. Let $T' = \{t_1, \ldots, t_k\}$ be such a subset of MT. Although all the transitions in T' could be ready to fire simultaneously, given a marking it may enable only a subset of the transitions in T'. Thus when calculating the guards of the do-loop, we must consider the firing of all its subsets. Without loss of generality, we study what happens to the subset $T' \setminus \{t_k\}$. There exists a marking for which all transitions in $T' \setminus \{t_k\}$ are enabled and t_k is not enabled.

The guard corresponding to the set $T' \setminus \{t_k\}$ in the do-loop is given by the conjunction $C_1 \wedge C_2$. Condition C_1 is simply the conjunction of the enabled conditions of the transitions in $T' \setminus \{t_k\}$. For condition C_2, firstly we calculate the set T'' of transitions which contains the transition t_k and

```
type SIPnet      = (Structure, Marking, Context)
type Structure = [Trans]
type Marking   = [Place] -- marked places
type Context   = [Guard]
data Trans     = Trans TransId [Place]
                       [Place] [Place] Guard [Place]
data Place      = Place PlaceId
type Guard      = Formula
type TransId    = Id
type PlaceId    = Id
type Id = String
```

Figure 6.20: Haskell data types to represent SIP-net models.

the transitions in the set $\bigcup_{A \in PEMAXS \wedge A \supset (T' \setminus \{t_k\})} \quad A \setminus (T' \setminus \{t_k\})$.

Secondly, we calculate the subset T''' of T'' consisting of the transitions not validated by any of the valuations which validate the transitions in $T' \setminus \{t_k\}$. Condition C_2 is then the conjunction of the negations of the enabled conditions with respect to transitions in T'''.

Performing the previous calculations to all subset T'' of MT, and all MT in $PEMAXS$, we obtain a do-loop modelling the structure of the SIP-net model, where the non-deterministic choices correspond to the choice of a valuation, for a given marking.

The ready conditions are the basic elements used to specify the behavioural properties of the SIP-net models. In the PROMELA specification, there is no explicit representation of the transitions' guards. Thus, the ready condition, for a transition t, is the disjunction of all guards in the do-loop, in which the enabled condition for t occurs. Notice that the ready condition of two or more transitions is not the conjunction of the corresponding ready conditions of those transitions, but the disjunction of the guards in the do-loop, in which enabled conditions for all considered transitions occur.

The Computer Application

In this section, the computer application that was created to generate the PROMELA specification for a given SIP-net model is presented. The aplication was written in Haskell. In order to describe an SIP-net model in Haskell we use the data type definition presented in Figure 6.20, that is we represent an SIP-net model in Haskell as a triple with the structure of the net, a marking and a context.

A marking of a net is represented in Haskell as the set of places which have one token. In Haskell a set is represented as an Haskell list. Notice

```
parsyn :: SIPnet
parsyn = [
 Trans "t0" [Place "p0"] [] [] (Var "x0") [Place "p2"],
 Trans "t1" [Place "p1"] [] [Palce "p0"]
                                (Var "x1") [Place "p3"],
 Trans "t2" [Place "p2"] [Place "p3"] []
                                (Var "x2") [Place "p0"],
 Trans "t3" [Place "p3"] [] [] (Var "x3") [Place "p1"] ]
prsm30 = [Place "p0",Place "p1"]
sparsyn= (parsyn,prsm30,[])
```

Figure 6.21: Haskell specification of the SIP-net model example.

that SIP-net models are safe nets, so a place has at most one token. This allows the use of a set to represent the marked places. A context is a set of guards, which is in Haskell a formula of propositional logic.

The structure of an SIP-net model is a set of transitions. Each transition has a label of identification (TransId), the set of pre-places, the set of the places linked through enabling arcs, the set of the places linked through inhibitor arcs, a guard, and the set of post-places, respectively. For example, the SIP-net model illustrated in Figure 6.15 has the Haskell representation presented in Figure 6.21.

Next we present functions included in the tool to implement the calculations described in the previous section. We also show its usage in the Haskell representation in Figure 6.21. The function enabled has the following signature:

```
enabled :: SIPnet -> [[Trans]]
```

Given an SIP-net model, the program calculates the maximal subsets of enabled transitions, which is denoted by $PEMAXS$ in the previous section. Applying this function to the SIP-net model, we obtain the following result:

```
*Ex_parsyn> enabled sparsyn
[[t1],[t2,t3],[t0,t3]]
```

For each subset of $PEMAXS$ we calculate the maximal subsets whose transitions could be simultaneously ready.

Given an SIP-net model and the set of enabled transitions, it is calculated the set of pairs, such that the first component has the transitions that can be ready. This ready condition is determined by the no satisfaction of the enabled condition of transition in each element of the second component.

Using the Application Using the previous results we can generate the PROMELA specification for the SIP-net model. Additionally, we generate the TL conditions to test the potential conflicts in the SIP-net model.

These conditions are based on their ready condition, already included in the PROMELA specification.

The toPROMELA function creates a String from an SIP-net model, and has the following signature:

```
1 toPROMELA :: SIPnet -> String
```

The result of applying toPROMELA function to a SIP-net model is a string with the PROMELA specification corresponding to the SIP-net model, that constitute the input to Spin tool to analyze the properties of the given SIP-net model. For the considered example we have the following PROMELA specification:

```
1  *Ex_parsyn> (putStr.toPROMELA) sparsyn
2  #define M 4
3  #define en_t0 (p[0] && !p[2])
4  #define en_t1 (p[1] && !p[3] && !p[0])
5  #define en_t2 (p[2] && !p[0] &&  p[3])
6  #define en_t3 (p[3] && !p[1])
7  #define fire_t0 p[0] = 0; p[2] = 1;
8  #define fire_t1 p[1] = 0; p[3] = 1;
9  #define fire_t2 p[2] = 0; p[0] = 1;
10 #define fire_t3 p[3] = 0; p[1] = 1;
11 /* Enabled Conditions for each transition */
12 #define ready_t0 (en_t0 && en_t3) || (en_t0)
13 #define ready_t1 (en_t1)
14 #define ready_t2 (en_t2 && en_t3) || (en_t2)
15 #define ready_t3 (en_t0 && en_t3) || (en_t3)
16 /* Enabled conditions for transitions
17 in a potential conflict. */
18 bool p[M];
19 init{ atomic{ p[0] = 1; p[1] = 1; p[2] = 0; p[3] = 0;  }
20 do
21   :: en_t0 && en_t3 -> atomic{ fire_t0; fire_t1; }
22   :: en_t2 && en_t3 -> atomic{ fire_t0; fire_t1; }
23   :: en_t0          -> atomic{ fire_t0; }
24   :: en_t1          -> atomic{ fire_t1;}
25   :: en_t2          -> atomic{ fire_t2;}
26   :: en_t3          -> atomic{ fire_t3;}
27 od }
```

In this PROMELA specification, for each transition, we have the definitions of: the enabled conditions (between lines 2 and 5), the fire actions (between lines 6 and 9), and the ready conditions (between lines 10 and 14).

These definitions use the variables representing the places of the net, which are declared in line 17. After that we have the init process. In line 18 the marking values are assigned to the places. The do-loop guards, lines between 21 and 25, represent the different sets of transitions which can fire simultaneously, and the corresponding action to fire the transitions.

The ready condition, for a transition t, is the disjunction of all guards in the do-loop, which have an occurrence of enabled condition for t. This can be seen in the previous example on the definition of ready conditions. The ready condition of two or more transitions is not the conjunction of the corresponding ready conditions to those transitions, but the disjunction of the guards in the do-loop, in which the enabled conditions for all considered transitions occurs. The ready conditions are the basic elements when specifying the behavioural properties of SIP-net models.

To check if a potential conflict among transitions constitute a behavioural conflict we need to evaluate the simultaneous enabling of these transitions, thus we need to have the enabling conditions of transitions in potential conflict. Thus we decide to define only the following conditions:

- the ready conditions for each transition (presented between lines 11 and 14); and

- the ready conditions for each set of transitions in potential conflict (in the example there are no potential conflicts, thus no enabling conditions for more than one transition is defined, but we put commentary on line 15).

Next, we present the specification and the verification of properties of the SIP-net models in the context of the PROMELA specification, using the TL formulas. The three SIP-net model properties considered here are: (1) behavioural conflicts freedom, (2) dead-lock freedom, and (3) liveness of its transitions.

To check the freedom of behavioural conflicts in SIP-net models, we use the TL formula with the $\Box$ operator and the negation of the ready condition corresponding to the set of transitions that are in conflict. If the transitions in potential conflict are the transitions t_i and t_j, the formula is the following: $\Box\ \neg ready\_ti\_tj$.

According to Definition 6.1.9 we can define in TL two levels of liveness (L_1 and L_4) in the context of the generated PROMELA specification. The operators $\Box$ and $\Diamond$ represent, respectively, the universal, and the existential quantification over the states of the system.

Given an SIP-net model N and $t \in T_N$. The TL formula to be verified in order to prove that: t *is* L_1-*Live* is checked by the formula $\Diamond\ ready\_t$; and t *is* L_4-*Live* is checked by the formula $\Box\Diamond\ ready\_t$. In this way, we can study the behavioural conflicts freedom, liveness of system's transitions and also the absence of dead-locks.

6.2 Elevator Controller System

The purpose of this section is to provide a consistent and complete description of the requirements for an elevator controller software system for low-rise building elevators. This case study is an adaptation from the description presented in the technical report [Blanco 2005].

6.2.1 General description

An elevator (also known as lift) is vertical transport vehicle to move people or goods between floors of a building. Typically an elevator is powered by electric motors that either drive traction cables and counterweight systems, or pump hydraulic fluid to raise a cylindrical piston. In this work we are considering elevator systems with electric motors to move essentially people.

The software of the elevator controller is responsible for the safe and efficient operation of elevator hardware components in order to allow elevator passengers travel between floors of the building.

An elevator controller needs to interact with the components present in the elevator system. We are assuming for this case study, an elevator controller that manages an elevator system with two cars in a building with six floors. Next we introduce the components in the elevator system we are considering for this case study.

In a building prepared to receive an elevator system there are the elevator shafts, and in each floor there are a connection between the floor hall and the elevator shaft. The elevator car is a compartment with some available free space to receive a set of passengers, and it can be moved by the motor connected with it, in the elevator shaft along the floors of the building. In this case study there are two elevator cars where the passengers travel from one floor to another one, along the six floors of the building.

Figure 6.22 depicts the context diagram for the elevator controller, where the main sensors and actuators present in the considered top-most entities (Floor and Car) are shown. We are considering a building with six floors. There are two cars where the passengers travel from one floor to another one. Each car has a door (called a Car Door) that opens when the car is stopped in a floor, to allow the passenger to enter or to exit the elevator car. Each Floor contains two Location Sensors, one for each car, to detect when the respective car is at or is arriving to the floor. In each Floor there are Hall Buttons to allow the passengers to call an elevator car indicating the direction (up or down) he wishes to travel. Obviously, the first and the last floors have only one button to select the unique direction that is

possible to travel (up for the first floor and down for the top-most floor). After being pushed a light in the Hall Button is turned on, and the button stay illuminated until the car that will service the passenger arrives. This light allows the waiting passengers to be informed that a car elevator in that direction has been already called. There is a Floor Door in each floor to protect the car's shaft, and the doors only open when the corresponding car is stopped at the floor.

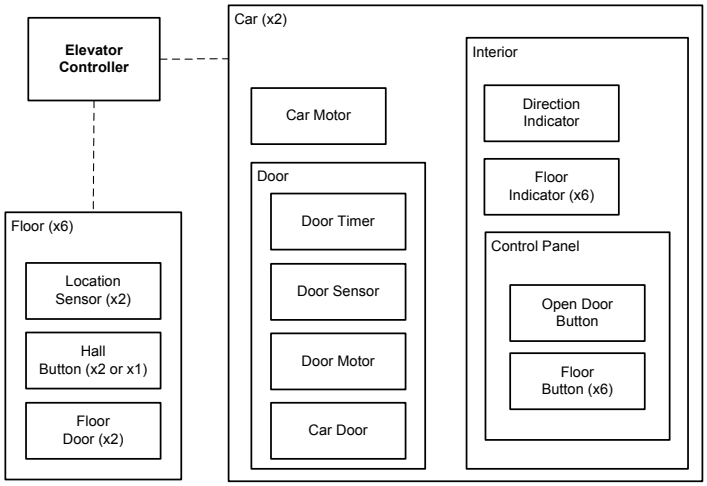


Figure 6.22: Context diagram for the elevator controller.

Each of the two Cars has one Car Motor, which is an electric motor connected with the elevator car to move up or down along the floors. Each car has a Door which has:

- a Car Door that opens when the car is stopped in a floor, to allow the passenger to enter or to exit the elevator car. It includes two sensors, to indicate either if the door is closed or totally opened;
- a Door Timer to define the period to wait before start closing the door, in order to automatically close the door after a given amount of time;
- a Door Sensor to detect if there is something obstructing the door during closure;
- a Door Motor to open or to close the car door;

When the elevator car is stopped at a floor the Car Door is mechanically linked with the Floor Door in the current floor, in this way the floor door opens or closes at the same time the car door is open or closed in that floor, without the direct intervention of the elevator controller.

Inside the car a passenger can find:

- a Floor Indicator that shows the current floor of the car,
- a Direction Indicator that shows the direction being followed by the car, and
- a Control Panel that contains an Open Door Button to open the doors and six Floor Buttons to select the destination floor.

6.2.2 Use cases descriptions

The behaviour of the elevator controller is triggered by the passengers' actions. A passenger can push the buttons in each floor to select the direction he wants to travel. When the passenger is inside a car he can select the floor he wants to travel, pushing the button with the number of the floor. Inside the car, there is also a button to open the door. The elevator controller has the responsibility of managing the movements of the cars in accordance to the requests of the passengers through pushing on the buttons.

Our approach proposes the usage of a use case diagram to depict the main functionalities provided by the system to its users. Figure 6.23 presents a use case diagram for the elevator controller system. The brief description of each use case in the Use case diagram of Figure 6.23 is the following:

UC1 - *Travel to Floor*: Passenger at a floor requests an elevator to travel in an intended direction;

UC2 - *Service Floor*: System moves an elevator car from an origin floor to a destination floor;

UC3 - *Open Door*: System opens the door of an elevator car;

UC4 - *Close Door*: System closes the door of an elevator car;

UC5 - *Announce Emergency*: Passenger announces an emergency in the elevator car where the passenger is in;

UC6 - *Ring Alarm Bell*: Alarm bell rings while bell button is being pressed by the passenger;

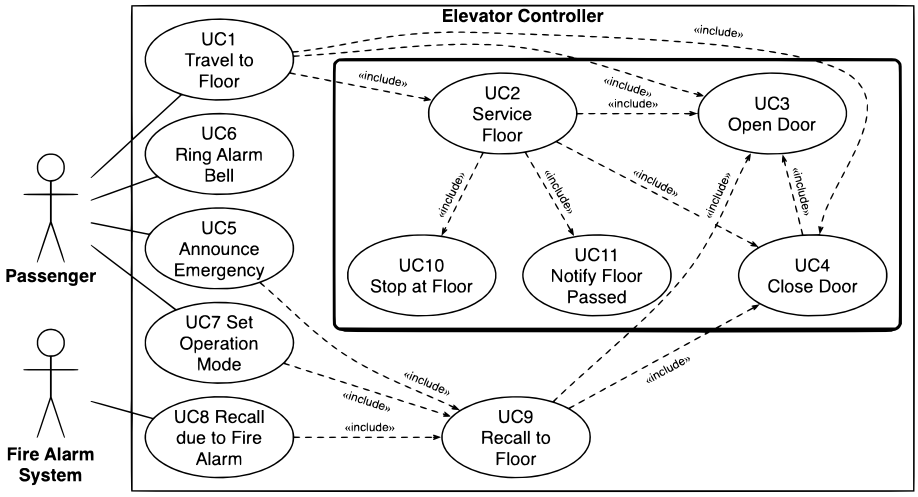


Figure 6.23: Use case diagram for the elevator controller system.

UC7 - *Set Operation Mode*: Passenger changes the operation mode of the elevator car;

UC8 - *Recall due to fire Alarm*: Elevator cars are recalled to a floor, determined by the fire alarm system in the building, when the fire alarm is activated;

UC9 - *Recall to Floor*: Elevator car is recalled to a floor;

UC10 - *Stop at Floor*: System stops the elevator car at a floor;

UC11 - *Notify Floor Passed*: System informs the passenger inside the elevator car that the car is just passed the floor.

To illustrate the usage of sequence diagrams to describe the scenarios present in a use case description, we start by detailing the use case UC2 called "Service Floor". This use case is responsible for moving an elevator car from an origin floor to a destination floor, by request of a passenger either in one of the building's hall or inside an elevator car.

Figure 6.24 depicts the sequence diagram with the main scenario of the "Service Floor" use case. This sequence diagram uses some high-level operators present in the UML 2.0, namely the opt and the loop operators. These operators permit the description of several scenarios in a unique sequence diagram.

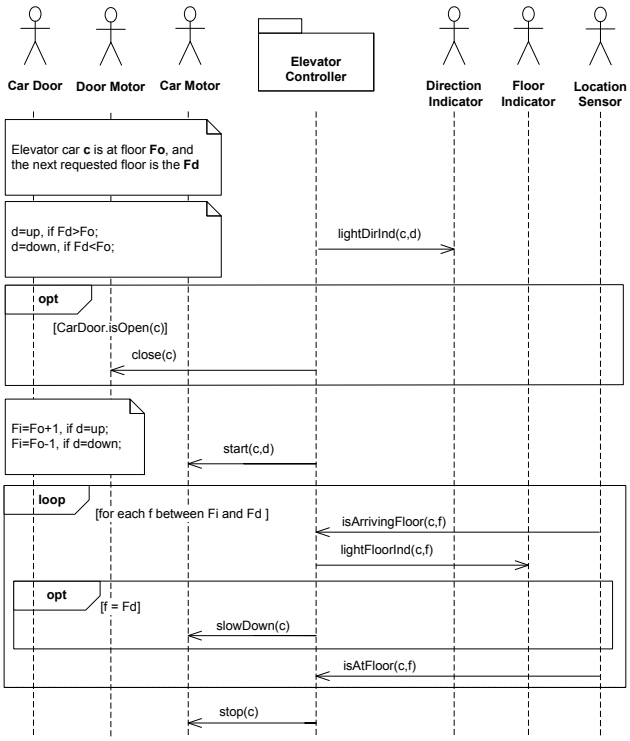


Figure 6.24: Sequence Diagram describing the "Service Floor" use case.

In order to abstract from the scenarios presented in the sequence diagram, there are along the left-hand side of the diagram some textual annotations where some variables being used in the messages (and guards) are informally declared. For example, the diagram in Figure 6.24 abstracts from the car, and the origin and destination floors. With the textual annotation *"Elevator car c is at Floor Fo, and the next requested floor is the Fd"*, we are declaring the variables c, F_o (origin floor) and F_d (destination floor), to be used as parameters for the scenario present in the diagram. With these variables, we are able to define variable d that represents the direction followed by the car, in the textual annotation "d=up, if Fd>Fo; d=down, if Fd<Fo". The textual annotation "Fi=Fo+1, if d=up; Fi=Fo-1, if d=down;" defines the variable F_i, representing the next floor from the origin floor.

The scenario presented in Figure 6.24 describes the following behaviour related to UC2.

1. The passenger in the current floor is notified about the direction the

car will take (message lightDirInd);

2. If the car door is open (high-level operator opt)
   (a) The car door is closed (message close)
3. The car is moved in direction to the destination floor (message start);
4. While the destination floor is not reached (high-level operator loop):
   (a) The location sensor informs the elevator controller that the car is arriving to the next floor (message isArrivingFloor);
   (b) The Floor Indicator corresponding to the next floor is activated (message lightFloorInd);
   (c) The Car Door must slow down its speed, when the next floor is the destination floor (message slowDown);
   (d) The location sensor informs the elevator controller that the car is at the next floor (message isAtFloor);
5. The car stops (message stop).

In this main scenario of the "Service Floor" use case, we are assuming that during its execution there are no other interactions of the passengers with the buttons. These situations could origin some other variations to this main scenario.

When a passenger requests the elevator to stop, it can be in an intermediate floor while it is moving. Figure 6.25 presents a sequence diagram that includes these variations, which is a sequence diagram obtained from the sequence diagram in Figure 6.24, adding the high-level operator loop.

Systems sends an elevator car to the floor from where the passenger is located, and carry the passenger to the destination floor. The main scenario for the UC1 *Travel to Floor* use case:

1. Passenger, who is at one of the floors in the building, pushes one of the hall buttons in the floor indicating the travel direction (up or down);
2. System selects one of the elevator cars to answer the request of the passenger;
3. System assigns the selected car to the request of the passenger;
4. System notifies the passenger about the assignment of a car to the request, by illuminating hall button pressed by the passenger;

5. System starts a service floor (use case *service floor*);
6. System notifies the passenger that the car has arrived to the floor;
7. System opens the door of the elevator car;
8. System turns on the direction indicator light inside the elevator car according to the direction that the passenger selected to travel;
9. Passenger enters in the elevator car;
10. Passenger pushes a floor button inside the elevator car to select the destination floor;
11. System records the request;
12. System notifies the passenger that the request was recorded, by illuminating the floor button that was pressed by the passenger;
13. System closes the door (use case *Close Door*);
14. System starts a service floor (use case *service floor*);
15. System notifies the passenger that the request was served, by illuminating the floor button that has been pressed by the passenger;
16. System opens the door of the elevator car;
17. Passenger exits the elevator car.

The main scenario for the use case "Travel to Floor" is described in the sequence diagram depicted in Figure 6.26.

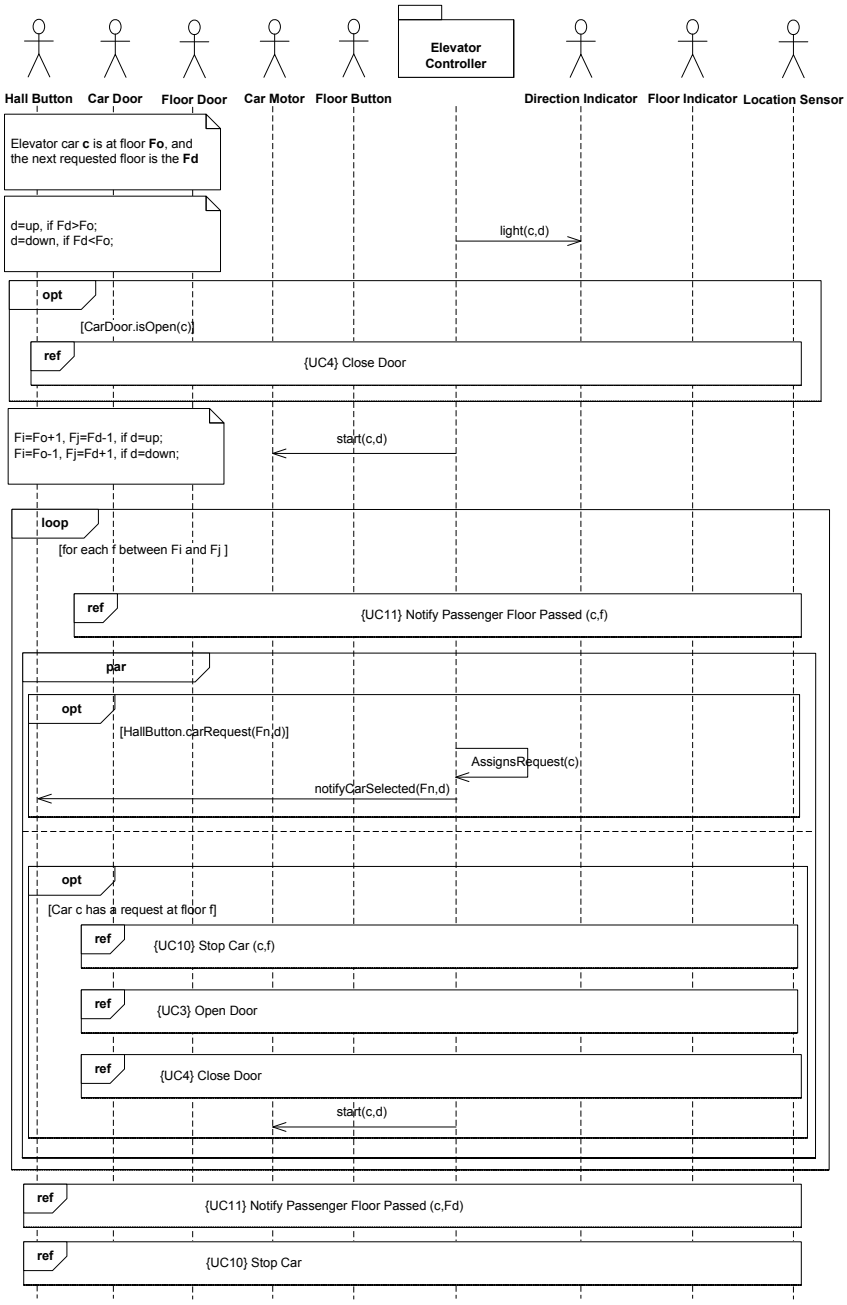


Figure 6.25: Sequence Diagram describing the main scenario of the use case UC2 “Service Floor”, and some of its variations.

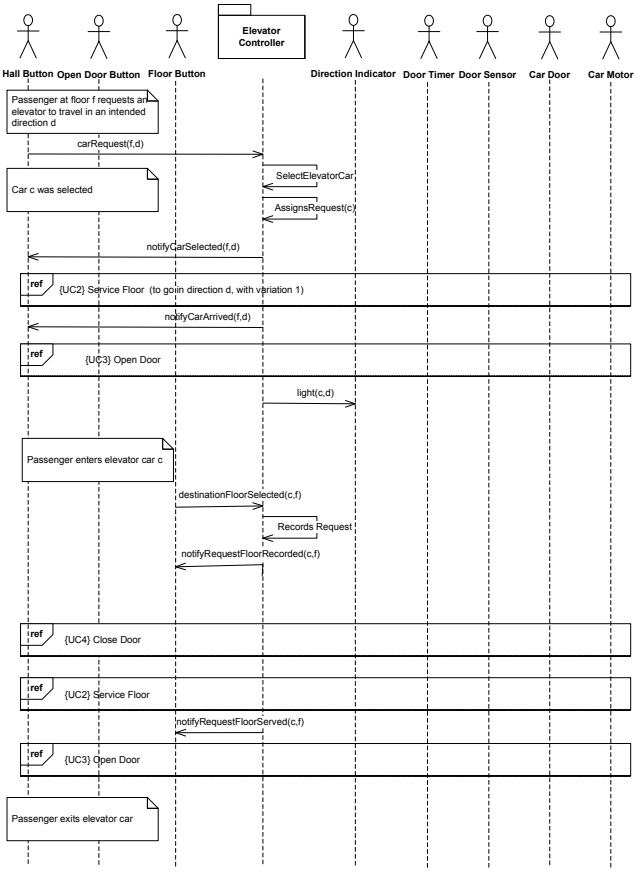


Figure 6.26: Sequence Diagram describing the main scenario of the use case UC1 “Travel to Floor”.

6.2.3 Expressing scenarios by a CPN model

In this subsection we present the CPN model to express some of the scenarios for the elevator controller system. We concentrate on the use case UC2 "Service Floor", whose sequence diagram is presented in Figure 6.25. Figure 6.27 presents the top-level CPN model obtained from this sequence diagram.

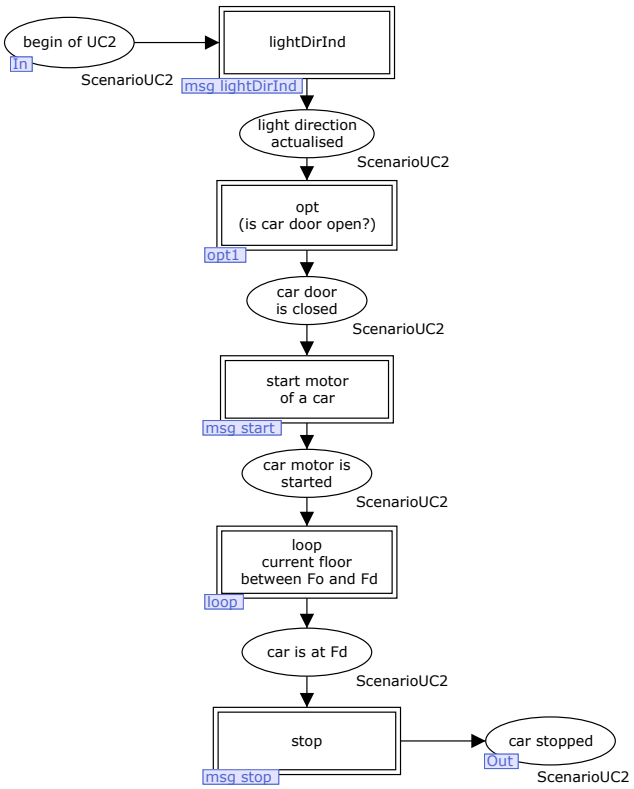


Figure 6.27: CPN model representing a sequence diagram for UC1 "Service Floor".

Figure 6.28 represents the initial part of the sequence diagram presented in Figure 6.26 for the use case UC1 "Travel to Floor".

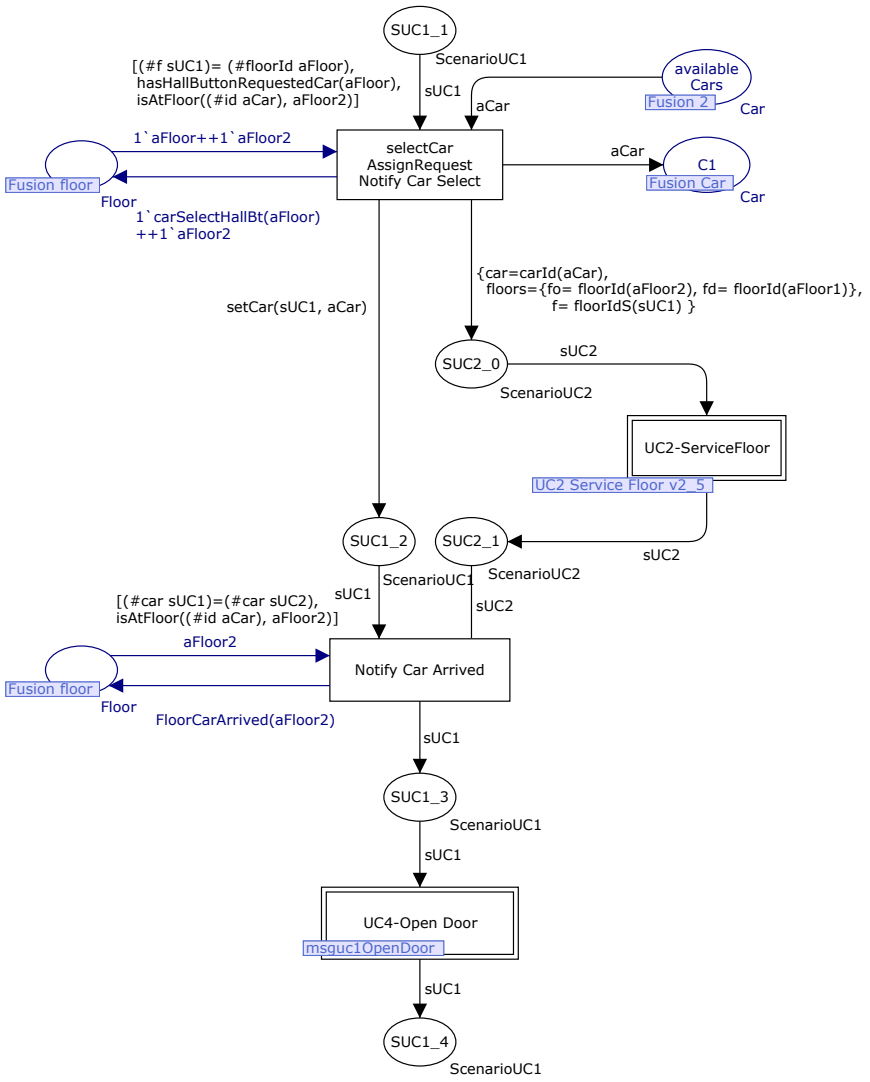


Figure 6.28: CPN model representing a sequence diagram for UC1 “Travel to Floor”.

6.2.4 Building an animation

This subsection presents the steps followed to build an animation of the elevator system and also the tools involved in the deployment of the animation. Only the functionalities related with use case UC2 “Service Floor” is

considered in this animation.

In the previous subsection, we present the elevator system consisting on an elevator controller managing two elevator cars in a building with six floors. In this subsection, we present the development of an animation layer for the elevator controller introduced in the previous section.

In this case study we consider the elements in the environment as the actors of the elevator controller system. The actors for the "Service Floor" use case are the ones that participate in the sequence diagram in Figure 6.24.

An elevator can be visually represented by a picture with the floors and the cars, where the cars can go up/down across the floors. While the cars are moving, the elevator controller must update the information shown in the panels inside each elevator car and attend the requests from the passengers.

For the elevator controller, only a subset of the entities of the environment are relevant for animation, since the passenger is not aware of (or does not interact with) all of them. This means that the animation layer only includes the relevant entities. In contrast, the CPN model does specify all the environment's entities. In the elevator controller, the passenger interacts only with the following six entities: Hall Button, Car Door, Direction Indicator, Floor Indicator, Open Door Button, and Floor Button.

Figure 6.29 presents a detailed version of the context diagram in Figure 6.22, and associates the graphical representation to some of the entities presented in the context diagram. A dashed arrow from one entity to the graphical icon means that the entity *is graphically represented by* the icon. If an entity as no that have no icons associated with it, it is not graphically represented in the animation.

Figure 6.30 shows a screenshot obtained from the animation of the elevator controller. On the left part of the figure, there is a representation of each floor with the buttons to call an elevator. We can also see the two elevator cars in Figure 6.30: the left-hand side elevator is at the second floor with its door open, and the right-hand side elevator is at the fifth floor with its door closed.

On the right part of the figure, the interior of both cars are depicted. Each car has a floor indicator, which has one light for each floor, and only the light of the current floor is on. There are also two lights, one to indicate the up direction and another one to indicate the down direction, showing the direction being followed by the car. Figure 6.30 shows that the left-hand side car has the light with the number two in yellow indicating that this light is on, and thus that the car is currently at the second floor. The left-hand side car currently has no direction, and the light direction of the right-hand side car indicates that the car is going down. One can use the associations

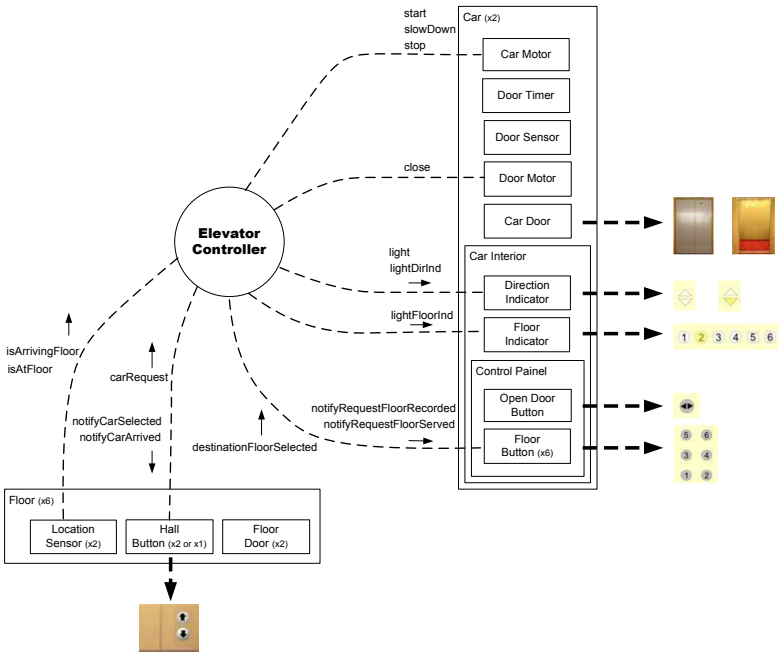


Figure 6.29: Diagram that associates graphical representations to some of the entities of the environment of the elevator controller.

between the entities of the elevator controller system and the graphical icons presented in Figure 6.29.

Scripting language

Next we show part of the Ruby script that generates the XML code for the animation of the Direction Indicator.

```
1 require 'builder'
2 require 'sbXMLgen.rb'
3 xmlBuilder = Builder::XmlMarkup.new(
4               :target => $stdout, :indent => 3)
5 xmlBuilder.instruct! :xml, :version => "1.0"
6 xmlBuilder.animation("width" =>"800", "height" => "600" ){
7   carIds = ["left","right"]
8   carInteriorCoord = {"left"  => Tuple.new(420,20),
9                       "right" => Tuple.new(610,20)}
```

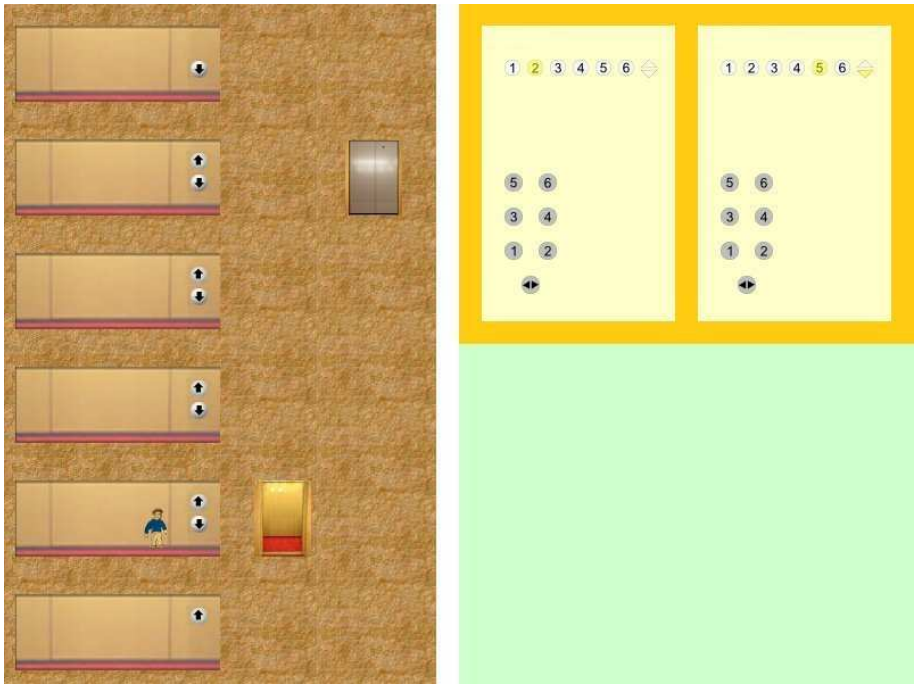


Figure 6.30: A screenshot of an animation for the elevator controller system.

```
10  lstBhLightDI = Hash.new()

11  lstCommands = Hash.new()
12  carIds.each{ |carId|
13    bhparams = Hash.new()
14    durQuick = 0.0001
15    xHide = -1000
16    xVisible =  carInteriorCoord[carId].getX+7*20
17    bhparams["show"] =
18        BehaviourParams.new(xHide,xVisible,durQuick)
19     bhparams["hide"] =
20         BehaviourParams.new(xVisible,xHide,durQuick)
21    directions = ["up", "down", "idle"]
22    movs = ["show", "hide"]
23    bh = Hash.new()
24    movs.each{ |m| bh[m] = Hash.new() }
25    lstBhLightDI[carId] =  {"up"=>[],
26                            "down"=>[], "idle"=>[]}
27    mkCmdStrDI= lambda{|mov, cId, dir|
```

```
28              "#{mov}lightDirInd(#{cId}Car,#{dir})"}

29    directions.each{ |d|
30      movs.each{ |mov|
31        cmdStrDI = mkCmdStrDI.call(mov,carId,d)
32         bh[mov][d] = Behaviour.new(cmdStrDI, "move",
33                        bhparams[mov], "xpto")
34         bh[mov][d].toXML(xmlBuilder)
35        lstBhLightDI[carId][d].push(
36            Tuple.new("x", cmdStrDI))
37        if (mov=="show") then
38           d_tmp = directions.dup
39           d_tmp.delete(d)
40           lstBhToStart = Array.new()
41            lstBhToStart.push(cmdStrDI)
42            d_tmp.each{ |od|
43            cmdStrDIod =
44              mkCmdStrDI.call("hide",carId,od)
45            lstBhToStart.push(cmdStrDIod)
46           }
47           cmdStrDIname =
48               mkCmdStrDI.call("",carId,d)
49           lstCommands[cmdStrDIname] =
50               lstBhToStart.dup
51        end
52        } }
53  }
54  lstCommands.each{|k,v| make_command(xmlBuilder,k,v) }
55  xmlBuilder.draw {
56   carIds.each{ |carId|
57     draw_DirectionIndicator(xmlBuilder,
58           carInteriorCoord[carId], lstBhLightDI[carId])}
59 }
```

In line 1, the package from the builder library to build XML tags is included. When we define an object as a Builder (see lines 3 and 4 where we define the variable xmlBuilder) we are able to generate a XML tag by using the name of the tag to be generated as a method for the object. For example in line 6 the following XML structure is created:

```
1 <animation height="600" width="800">
2 ...
3 </animation>
```

The tag animation includes the XML code generated by the lines 7-47 of the Ruby script.

In line 6, the variable carIds is defined as an array with the car identifiers. It is possible to create an iteration over this variable, as is shown in line 10. This permits an easy integration of new similar cars in the animation, because we have a dynamic structure based on the elements into the array carIds.

6.3 Check-in System

This section describes a case study used to exemplify the application of our approach. The case study consists on a check-in system in an international airport [Robertson and Robertson 2006]. The results presented in this section were published in [Ribeiro and Fernandes 2009].

6.3.1 General description

The main stakeholder in this system is the so called check-in agent that interacts with the passenger in order to execute the check-in of the passenger. An informal description of the checking in of a passenger is given by one of the check-in agents in the following paragraphs [Robertson and Robertson 2006].

> *"I call the next customer in line when he gets to my desk, I ask for a ticket. If the passenger is using an e-ticket, I need the booking record locator: Most of the passengers are not organized enough to have it written down, so I ask them their name and the flight they are on. Most people do not know the flight number, so I usually ask for his destination. They must know that!"*
>
> *"I make sure I have the right passenger and the right flight. It would be pretty embarrassing to give away someone else's seat or to send a passenger to the wrong destination. Anyway, somehow I locate the passengers' flight record in the computer: If he has not already given it to me, I ask for the passenger's passport. I check that the picture looks like the passenger and that the passport is still valid."*
>
> *"If there is no frequent-flyer number showing against the booking, I ask the passenger if he belongs to our mileage scheme. Either he hands me the plastic card with the FF number, or I ask him and if he wishes to join I give him the sign-up form. We can put temporary FF numbers against the flight record so the passenger is credited for that trip."*
>
> *"If the computer has not already assigned a seat, I find one. This usually means I ask if the passenger prefers a window or an aisle seat, or, if the plane is already almost full, I tell him what I have available. Of course, if the computer has assigned one, I always ask if it is okay. Somehow we settle on a seat and I confirm it with the computer system. I can print the boarding pass at this stage, but I usually do the bags first."*

> *"I ask how many bags the passenger is checking in and, at the same time, verify that he is not exceeding the carry-on limit. Some people are unbelievable with what they want to carry into fairly space-restricted aircraft cabin. I ask the security questions about the bags and get the passenger's responses. I printout the bag tags and securely attach them to the bags, and then I send the bags on their way down the conveyor belt."*
>
> *"Next I print the boarding pass. This means that I have everything done as far the computer is concerned. But there is one more thing to do: I have to make sure that everything agrees with passenger's understanding. I read out form the boarding pass where he is going, what time the flight is, and what time it will board. I also read out how many bags have been checked and confirm that their destination matches the passenger's destination. I hand over the documents, and wish the passenger a good flight."*

Based on this description one can sketch out the scenario, that contains the steps capturing the normal path through the descrition. A possible sequence of steps performed by the check-in agent during the main scenario of the check-in passenger business use case is the following [Robertson and Robertson 2006]:

1. Get the passenger's ticket or record locator;
2. Confirm that this is the right passenger, flight, and destination;
3. Check that the passport is valid and belongs to the passenger;
4. Record the frequent-flyer (FF) number;
5. Find a seat;
6. Ask security questions;
7. Check the baggage onto the flight;
8. Print and hand over the boarding pass and bag tags;
9. Wish the passenger a pleasant flight.

The main scenario is modelled by the UML2 sequence diagram illustrated in Figure 6.31, where we can observe a parallel high-level operator that represents the fact that steps 4 and 5 ("record the FF number" and "find a seat" respectively) can be done in any order.

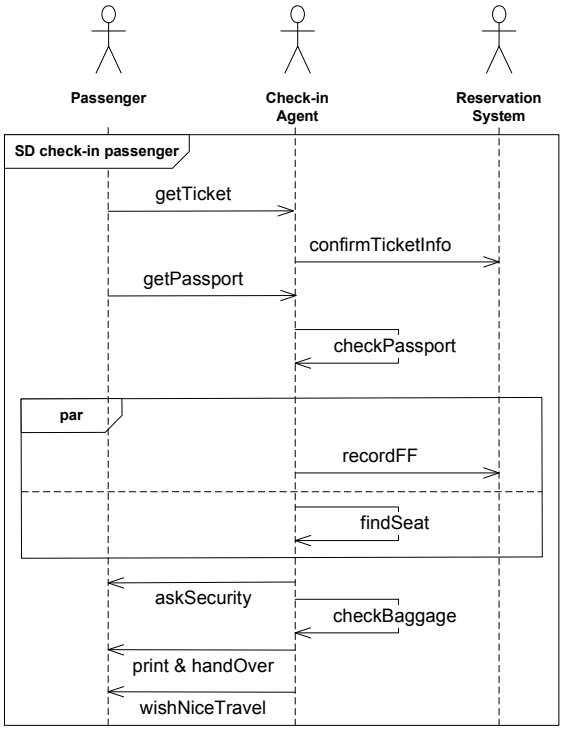


Figure 6.31: Sequence diagram for the main scenario of the "check-in passenger" business use case.

Each step in the scenario is represented by a message in the sequence diagram. The sender and the receiver of a message are extracted from the step's description, and the order between these messages is the same as the order introduced by the numbers of the corresponding steps in the textual descriptions.

Some alternative and exception scenarios can be added to the main scenario. For example, three alternatives for step 4 were identified:

A4.1 Allow the FF number to be changed to that of a partner airline;

A4.2 Allow the FF number to be changed to that of a family member; or

A4.3 Allow the mileage of the flight to be donated to a charity of the passenger's choice.

The "A4" prefix in the previous items are used to indicate that the items constitute an alternative for step 4 in the main scenario, and each alternative

item is also enumerated.

In the case that the passenger has an invalid passport, an exception to the main scenario is introduced and the execution of the scenario must be ended.

The behaviour of the main scenario, with the alternatives listed above together with the exception for the invalid passport could be expressed by the sequence diagram in Figure 6.32. Notice that this sequence diagram describes a set of elementary scenarios, since some high-level operators are used.

6.3.2 Expressing scenarios by a CPN model

To illustrate our approach we show how it can be applied to the "check-in passenger" business use case. This subsection explains the transformation of UML2 sequence diagrams, describing scenarios, into a CPN model. After explaining how to obtain the CPN model for the main scenario of the use case, we show how to add alternative and exception scenarios, and how to enrich the CPN model with some possible (unexpected) behaviours performed by the passengers. The interested reader is referred to Chapter 4, where the modelling guidelines that were applied to iteratively obtain a CPN model from a set of sequence diagrams are presented.

Expressing the main scenario

The main scenario of the use case is described by the sequence diagram in Figure 6.31 and it gives rise to the CPN model presented in Figure 6.33.

The initial state for the considered scenario is modelled by the tokens inside the two places at the top of Figure 6.33. The place ready to check-in passengers contains the tokens that represent the passengers that are ready to proceed with the check-in, and the place available agents contains the tokens that represent the available check-in agents. These two places are used as input places to the transition getTicket, which represents the step in the scenario where the passenger gives the ticket to the agent. After this trigger, the passenger and the agent must proceed together to complete the steps of the scenario. There are places between transitions to save the information related to the considered passenger and agent along the scenario execution and to preserve the order between the transitions (according to the order given in the sequence diagram). The input arcs for getTicket have the inscriptions p and a that are variables representing a passenger and an agent, respectively.

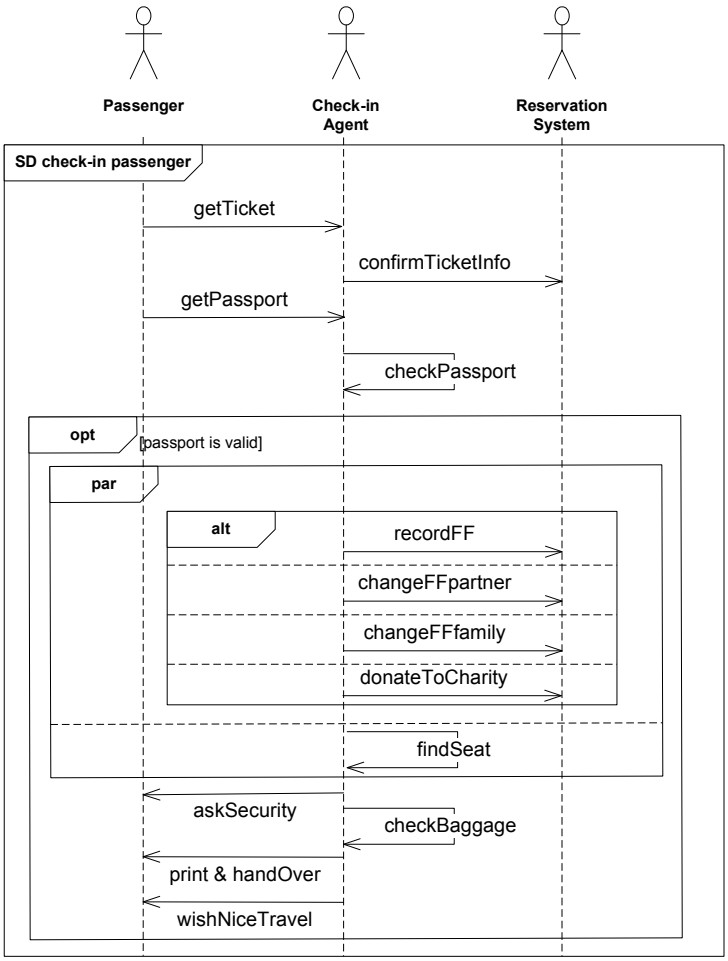


Figure 6.32: Sequence diagram with alternative scenarios of and exceptions to the "check-in passenger" business use case.

The execution of each instance of the scenario terminates when the considered passenger is checked-out, which implies that the agent is available again to start a new check-in procedure. The place checked-out passengers at the bottom of Figure 6.33 is used to contain the tokens of the passengers that were successfully checked-out. The details about the colour sets used in the places of the CPN model can be found below.

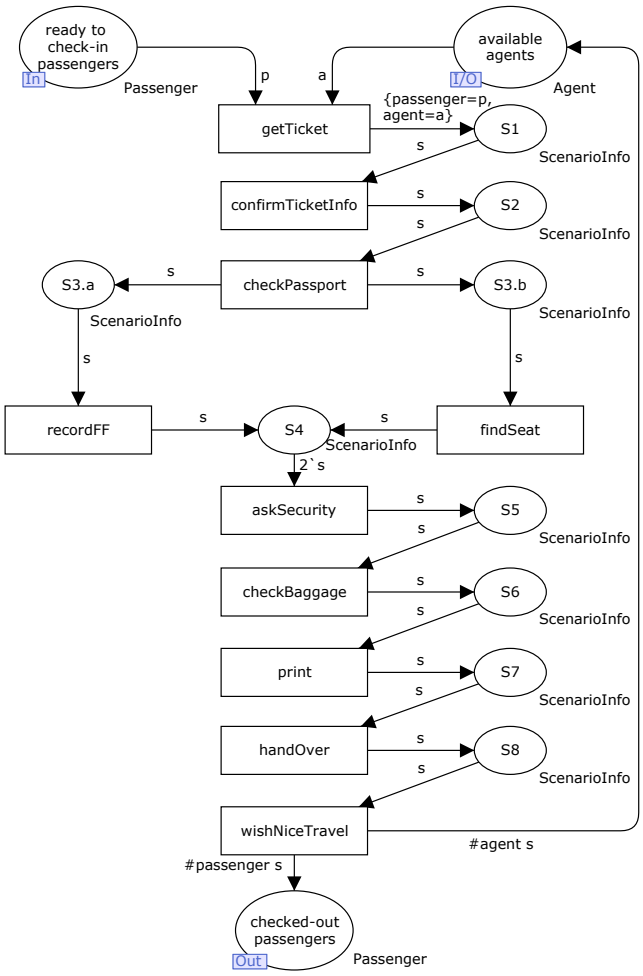


Figure 6.33: CPN module to represent the main scenario of the "check-in passenger" business use case.

Adding alternative and exception scenarios

The alternatives and exceptions are integrated in the CPN model obtained for the main scenario, in the following way. As explained above, the exception introduced by the usage of an invalid passport, and the alternatives introduced by the possibility to save the flight in a FF number of a partner airline, to save the flight in a FF number of a family member, or to donate the mileage to a charity.

The resulting behaviour is described by the sequence diagram in Figure 6.32, where we can find *opt* and *alt* high-level operators. The CPN module that reflects these alternatives and exceptions is based on the version for the main scenario in Figure 6.33.

Figure 6.34 presents the resulting CPN module obtained after considering the sequence diagram in Figure 6.32, where the transitions recordFF and checkPassport are now substitution transitions so that they represent the behaviour introduced by the high-level operators. The substitution transition recordFF is connected to the CPN module presented in Figure 6.35, where there is an alternative execution between the four presented transitions. Notice that in the common input place S3.a there is only one token for each scenario instance (in this case a scenario instance is represented by a pair of a passenger and a check-in agent). The substitution transition checkPassport is connected to the module in Figure 6.36, where there are two alternative transitions, one for the case when the passport is valid, and another when the passport is invalid. The use of an invalid passport is considered an exception, implying that the execution of the scenario must terminate. The end of the scenario execution moves the token representing the passenger participating in the scenario to the place idle passengers (which represents passengers not ready or willing to check-in), and the token representing the check-in agent to the place available agents.

Adding the behaviour of actors to the CPN model

We assume, as explained in [Jackson 2000], that passengers have free will and might behave in unexpected ways. Check-in agents also have free will, but they are more biddable, so they are expected to behave in a more constrained way. Therefore, in this case we enrich our CPN models with alternative scenarios triggered by possible abnormal or unexpected behaviours of the passengers, which may include, for example, ignorance, insubordination or malevolence. This is an important issue to make the CPN models more useful for animation and validation purposes, since they address a richer set of scenarios of usage.

In the context of the "check-in passenger" business use case, three main states were identified for passengers. In the main scenario we have identified the state (to begin the scenario) when the passenger is ready to check-in, and another state to finish the scenario when the passenger is checked-out. These two states are represented in the obtained CPN module in Figure 6.33 by the places ready to check-in passengers (at the top of the figure) and checked-out passengers (at the bottom of the figure). When considering alternatives and

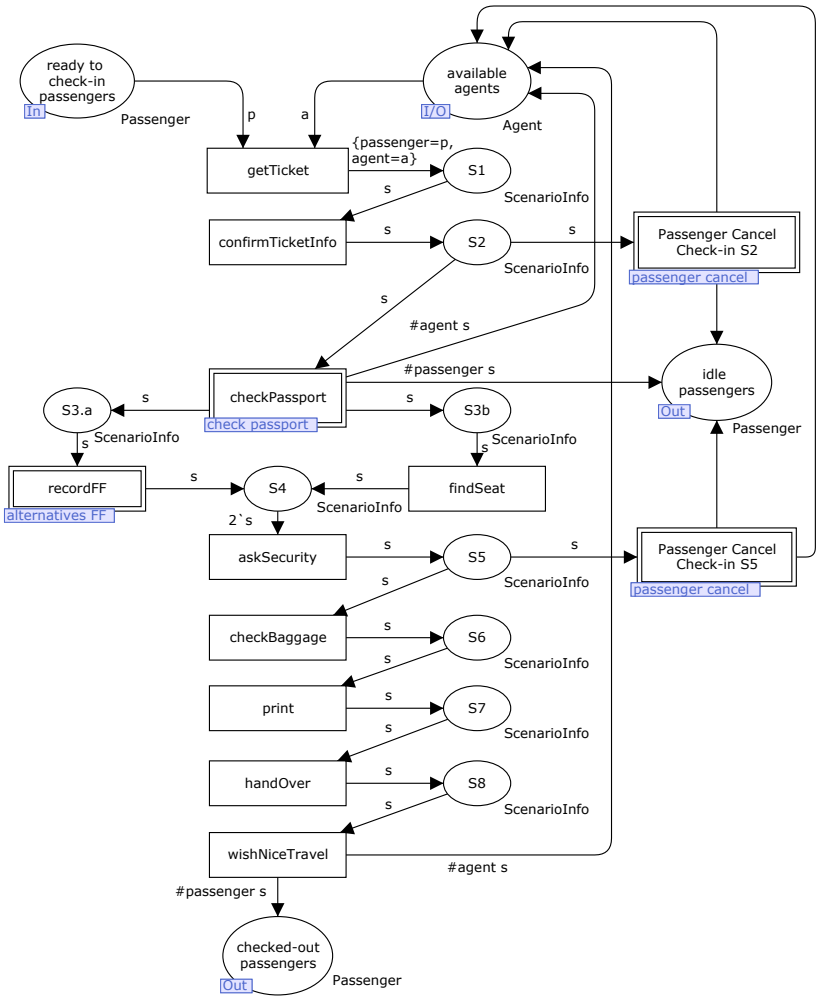


Figure 6.34: CPN module to express an alternative of and an exception to the main scenario of the "check-in passenger" business use case.

exceptions to the main scenario a third state was identified to capture the fact that the passenger is idle. For example, when the agent checks that the passenger's passport is invalid the check-in scenario ends and this passenger is considered to be idle. The CPN module presented in Figure 6.34 has a place idle passengers to contain the tokens corresponding to passengers that are idle.

To capture some of the abnormal behaviours introduced during the ex-

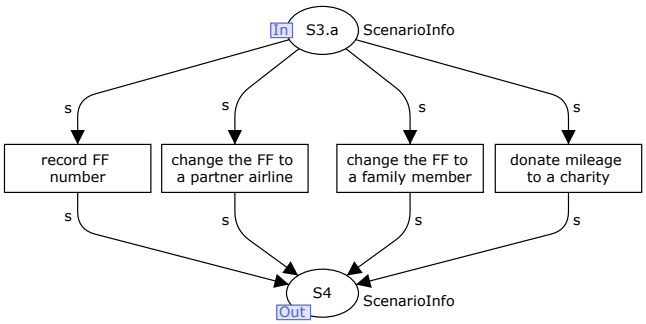


Figure 6.35: The altenativesFF CPN module.

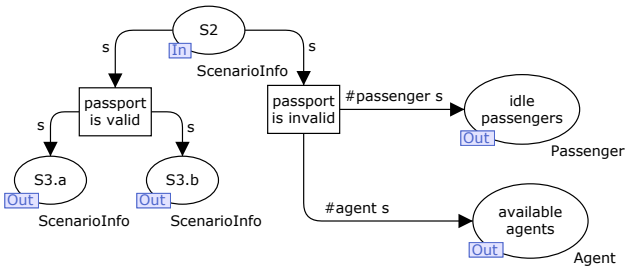


Figure 6.36: The check passport CPN module.

ecution of a scenario we can enrich the CPN model with a specific module. The CPN module that expresses the considered behaviours of the passenger is presented in Figure 6.37. We consider that an idle passenger can go for

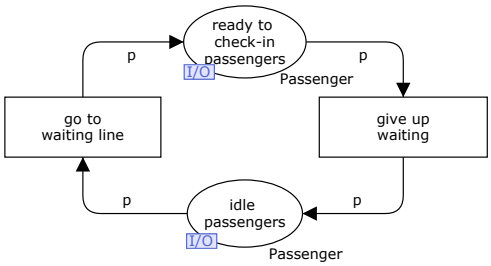


Figure 6.37: The passenger behaviour CPN module.

the waiting line for proceeding with the check-in (transition go to waiting line in Figure 6.37). A passenger in the waiting line can decide to abandon

the area, becoming idle again (transition give up waiting in Figure 6.37).

We model the cancellation of the check-in process by the passenger after the confirmation of the ticket information, or after the passenger being asked about the security issues (these two execution points are represented by the places S2 and S5 in Figure 6.34). The substitution transitions "Passenger Cancel Check-in S2" and "Passenger Cancel Check-in S5" in Figure 6.34 introduce the cancellation in the two corresponding execution points. After the cancellation, the passenger can start again from the beginning of check-in process.

Generalizing the CPN Model

In this subsection, we explain how a CPN model can be generalized in order to allow more use cases and their scenarios to be executed in parallel. Therefore we describe the role of colour sets used in the places of the CPN models and how the usage of multiple tokens in the places of a CPN model allows the parallel and concurrent execution of scenarios.

The top-most CPN module is presented in Figure 6.38. This module

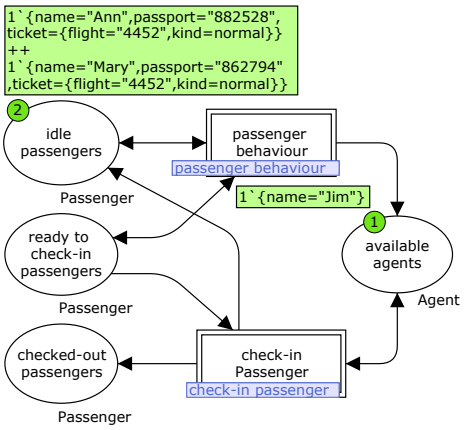


Figure 6.38: The top-most CPN module.

integrates, using substitution transitions, all the CPN modules presented previously, namely the ones for the main scenario of the "check-in passenger" business use case in Figure 6.34 and its exceptions and alternatives, and for the behaviour of passengers in Figure 6.37.

We have defined a colour set to represent each actor of the system that appears in the use case. The passengers and the check-in agents are the

actors for this system, and their colour sets are listed in Figure 6.39.

For the passenger, we consider that it is relevant to have her name, her passport number, and her ticket information. For the agents, only the name is used. As explained above the check-in of a given passenger is triggered when she shows her ticket to one of the available check-in agents.

To save the information about the scenario being executed, we use a colour set based on a record with the information of a passenger and an agent, called ScenarioInfo (see Figure 6.39).

The CPN module in Figure 6.38 has two tokens in the place idle passengers and one token in the place available agents. Near to each of these two places, there is a rectangle where the values of its tokens are specified.

```
1 colset Passenger =
2          record name: STRING*
3                 passport: INT*
4                 ticket: TICKET;
5 colset Agent =
6          record name: STRING;
7 colset ScenarioInfo =
8          record passenger: Passenger*
9                 agent: Agent;
```

Figure 6.39: Declaration of the colour sets.

In the specific case shown in Figure 6.38, no more than one passenger can simultaneously do the check-in, because there is only one available agent (called Jim). The introduction of an additional agent (adding one token with his identification to the place available agents) potentially allows two passengers to concurrently perform the check-in operation.

Consider more use cases

In this subsection we exemplify how to add more CPN modules representing different use cases to the CPN model. We consider the use case that permits a passenger to use the services associated with her FF card. This use case is executed by a different type of agents.

Assuming that there is a CPN module called use FF card to represent the scenarios of this use case, that module is integrated in the top-most module by adding a substitution transition as depicted in Figure 6.40. Therefore, the CPN model in Figure 6.40 permits instances of different use cases to be executed (an consequently animated) in parallel.

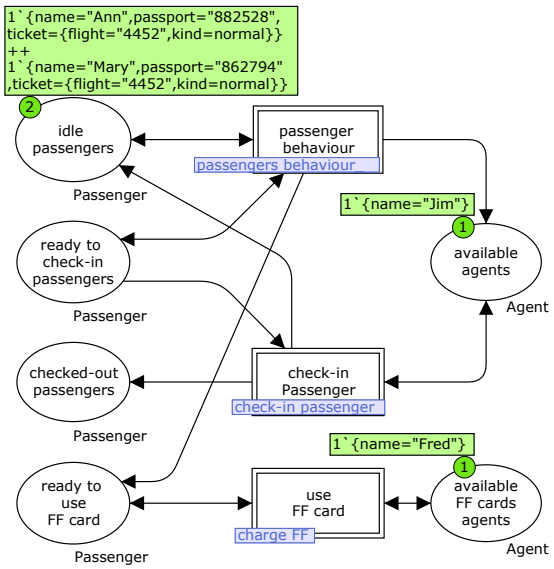


Figure 6.40: The top-most CPN module, with an additional use case.

6.3.3 Building an animation

This subsection presents an animation of the CPN models presented above for some scenarios of the check-in system. Figure 6.41 presents a screen-shot of an animation for the check-in passenger system. There is an icon of a human figure to represent each passenger using the system, who are identified by a number next to him. The check-in desks are depicted by a rectangle, with a human figure inside it to represent the respective agent. On the bottom of each check-in desk icon there is a number to identify it. On the top of each check-in desk icon there is an icon to depict what the agent is doing. For example, the check-in agent identified by the number 6 is now printing the boarding pass and bag tags for the passenger number 5. The passenger number 1 is at check-in 1, and the check-in agent has just validated the ticket. On the bottom of the animation there are three icons to represent the entrance door (on the left), the waiting room (on the middle), and the checkout door (on the right).

This animation is parameterized by the number of passengers, and the number of check-in desks, to be used on the animation. Depending on the initial number of elements to be included in the animation, the elements are distributed along the available space.

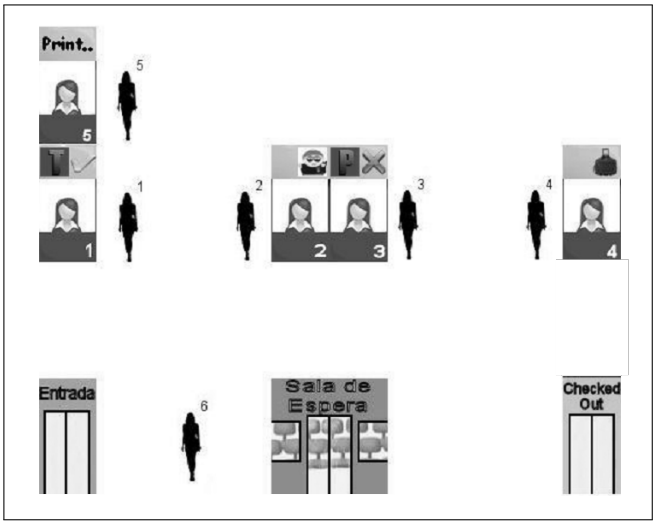


Figure 6.41: A screenshot of an animation for the check-in system.

6.4 Discussion

This chapter describes the three case studies considered in this thesis. Firstly, the reactor system case study has been considered. The reactor system case study was started by modelling it using a synchronous and interpreted Petri net variant, as presented in Subsection 6.1.5. This study contributed to detect a set of limitations in the usage of this initial approach, in particular when conducting the verification of properties of the model, we have detected the dependence of the results on the choice of the correct values for the variables that represent the elements of the system environment. This choice is in some contexts not free, which may imply that more scenarios is being considered than the need ones. In fact making a previous analysis of the behaviour of the environment may reduce its model complexity, and consequently facilitates the comprehension and also the verification of the properties of the model. After this exercise, we have started modelling the reactor system using the CPN modelling language, where the environment is also described. To accomplish this, the rules to transform a sequence diagram were applied to obtain the CPN model. In order to better understand the system behaviour we have created a graphical animation to validate the requirements through the execution of the CPN model.

Secondly, an elevator controller system has been considered. This system

is a reactive system where the humans have an important role on its operation, because its goal is to serve, human users, who are called passengers. Although important, the passengers are not themselves part of the system, since they only interact with system through the existing actuators and sensors. Various versions of the CPN models have been created (based on the scenarios descriptions), and we have studied how to easily parametrize the models in terms of the number of passengers using the elevator system An animation centered on the use case to travel a passenger between floors has been considered. This animation has been used in some contexts to illustrate our approach for the community.

Thirdly, the system to check-in passengers in an airport has been considered. Due to the abstraction level that we have considered for this system, we have included behaviours produced by humans (i.e. the check-in agents) as being part of the system. We have explored the possibility for abnormal behaviours performed by the human users of the system.

Chapter 7

Related Work

Summary

This chapter presents the state-of-the-art of the research fields considered in this work. The chapter starts with the analysis of some efforts on turning the use case descriptions into a more formal representations. After that, we consider some methods to directly generate a state-based model from a scenario-based descriptions, where there are no explicit reference to states of the system. The use of Petri nets as a medium to a synthesise scenarios describing a system behaviour is analysed. This chapter ends with the description of some efforts that have been presented to improve the way that formal specifications are presented to the users, in particular when these specifications represent software requirements.

Contents

7.1 Formalization of Use Case Descriptions 156
7.2 Generation of State-based Models from Scenarios 157
7.3 Synthesis of Petri Net Models from Scenarios . 159
7.4 Animating Formal Specifications of Requirements 166

7.1 Formalization of Use Case Descriptions

Several researchers have tried to formalize the informal aspects of use cases. While earlier approaches focused primarily on developing the formal semantics of use cases, recent proposals put more emphasis on developing techniques to integrate and perform analysis on a set of use cases.

As an example of the former, Hsia et al. [1994] use a BNF-like grammar to formally describe use cases. A graphical and tree-like representation, called the conceptual state machine, is used. This approach is effective when applied to a small number of relatively simple use cases. Specification based on multiple viewpoints can, in principle, be supported by developing a more complex grammar. However, this is likely to be too cumbersome to be useful on industrial applications where frequent changes to requirements are expected to occur and where iterative and incremental requirements elicitation techniques are needed.

Andersson and Bergstand [1995] adopt Message Sequence Charts (MSCs), which are frequently used to state the requirements for telecommunications software. There are several features in the MSC standard aimed at enhancing the expressiveness of individual MSCs. Examples include constructs to specify the conditional, iterative, or concurrent execution of MSC sections. Anticipated exceptions and required system responses can be specified, too. The authors argue that an MSC-based approach has advantages over the grammar-based approach in terms of scalability and understandability.

Dano et al. [1997] suggest use cases to be collected and described by tables, to facilitate the communication between the analyst and the domain expert. Later, through mapping rules, Petri Nets are built from the tables to formalize the requirements. The approach is used for producing object-oriented requirements specifications, based on structural models and focuses on deriving intra-object behavioural models.

In contrast, several researchers have proposed analysis techniques for a set of interacting use cases. Glinz [1995] uses statecharts to represent use cases. The relationships among use cases are represented using one of the following constructs: sequence, alternation, iteration, or concurrency. This approach assumes a disjointedness among the use cases and does not support overlapping scenarios where the same event sequences appear in multiple use cases. That is, when overlapping scenarios are later identified, existing use cases (and, consequently, corresponding statecharts) have to be modified to maintain the disjointedness. For example, statecharts connected by a sequence relation may need to be further decomposed into more detailed statecharts connected by sequence as well as by alternation constructs.

This approach does not satisfy the insensitivity property unless all of the overlapping scenarios are known in advance. This "forced" (and potentially unnatural) modification of statecharts does not support traceability.

Other approaches for analyzing dependencies include timed automata [Some et al. 1996], finite state automata [Lustman 1997], and MSCs [Leue and Ladkin 1996]. They are adequate for describing use cases individually and can even analyze the interactions among a small number of use cases. However, the larger the number of use cases there are to analyze, the more difficult it becomes to grasp and analyze the global system behaviours since the brute-force approach of considering all possible combinations quickly leads to the state explosion problem.

A formal syntax and semantics based in Petri nets for describing use cases in order to overcome the limitations of the use cases is proposed in [Lee et al. 1998].

7.2 Generation of State-based Models from Scenarios

The main focus of our work is on the transformation of models of behavioural scenarios into state-based models. In particular we are interested in the translation of sequence diagrams into CPN models. Next, we describe several approaches which were already proposed to combine the usage of these two types of models.

Harel [2001] proposes the usage of scenario-based programming, through UML use cases together with play-in scenariosThe play-in scenarios make it possible to go from a high-level user-friendly requirements capture method, via a rich language for describing message sequencing, to a full model of the system, and from there to the final implementation.

Harel and Marelly [2003b] developed an engine, called play-in/play-out, to obtain Live Sequence Charts (LSCs) from the requirements. The play-in/play-out engine allows end users to capture LSCs by playing with a mock-up of the final system's user interface. The user clicks on the buttons, fills in text fields, and simulates the system reaction. The user is said to play-in the scenarios. Afterwards, scenarios that have been played-in can be played-out, i.e., interpreted by the play-out engine. In order to do so, the engine monitors input events, such as buttons being clicked, text fields being filled in, and so on, and tracks these events in all LSCs of the specification. This approach actually simulates a minimal implementation of the future system. Harel and Marelly [2003a] present an approach called "*Come Let's*

Play" to support the idea that scenarios can do much more than only describing a single execution or folded set of executions of a system, because they can define the behaviour of a system. Their play engine [Harel et al. 2003] demonstrates that a system can be fully defined in terms of scenarios without providing a model of the underlying components, and that the operational state-based behaviour of the components can be synthesized from the scenarios. In this way, one can see that scenarios are the system or, at least, fully defined the system's behaviour.

In many situations, however, we do not need to define the full system behaviour from scratch. Instead, major parts of the system are either reused or consist of pre-fabricated components with known behaviour. Thus, scenarios, as the means to describe the expected behaviour of the new parts of the system, have to be incorporated with existing, usually state-based, behavioural models.

Another extension in LSC of the MSC standard is the notion of a synchronous message exchange. The distinction between possible and necessary behaviour is possible on single events, on parts of an LSC or an complete LSCs. Additionally, the inversion of this distinction allows the specification of forbidden behaviour, i.e., traces that the system must not execute.

A major obstacle to the idea of synthesizing statecharts from LSCs is the high computational complexity of the synthesis algorithms, that does not allow the approach to be applied to large systems. Additional problems are more methodological, related to the level of detail required in the scenarios to allow meaningful synthesis, the problem of ensuring that the LSC requirements are exactly what the user intended, and a lack of tool support and integration with existing development approaches.

Krüger et al. [1999] suggest the usage of MSCs for scenario-based specifications of component behaviour, especially during the requirements capture phase of the software process. They discuss how to schematically derive statecharts from MSCs, in order to have a seamless development process.

Whittle and Schumann [2000] propose an algorithm to automatically generate UML statecharts from a set of UML sequence diagrams. This work also presents the usage of the algorithm for a real application. Their main conclusion is that it is possible to generate code mostly in an automatic way from scenario-based specifications. This algorithm is applied in [Whittle et al. 2003, 2005] to the weather control logic subsystem of Center TRACON Automation system which is under development at NASA Ames Research Center. The result of this study is that it is possible to use scenario-to-state machine algorithms to reliably develop models of a complex and practical system.

Soares and Vrancken [2008] suggest a meta modelling approach to transform UML2.0 sequence diagram to Petri Nets in order to represent the real-time constraints and the shared resources, characteristics that are not addressed by the sequence diagrams.

Störrle [1999] presents an approach to translate a sequence diagram into a PN model, where a linear place-transition path is used to represent the behaviour of each object, associating a message to the events of sending or receiving messages. The two transitions that represent the send and receive of a message are either connected by a communication place if the message is asynchronous, or gathered into a single transition if the message is synchronous.

7.3 Synthesis of Petri Net Models from Scenarios

There are also works on the synthesis of Petri nets from scenario-based specification. Juhás et al. [2005] present a polynomial algorithm to decide if a scenario, specified as a Labelled Partial Order, is executable in a given Place/Transition PN. The algorithm preserves the given amount of concurrency and does not add causality. In case the scenario is indeed executable in the PN, the algorithm computes a process net that respects the concurrency expressed by the scenario. Although quite useful, this technique is not yet available for high-level PNs (such as Object-oriented PNs, CPNs, or Reference nets), and to validate the scenario the user must simulate the obtained process net, where the concepts of the problem domain are not clearly represented as in the created animations used in our approach.

Amorim et al. [2005] introduce an informal methodology to map LSCs into CPNs for allowing properties of the system to be verified and analysed. They do not consider the validation of the gathered behavioural scenarios, but only their verification, namely to detect some inconsistencies between them.

Eichner et al. [2005] present a formal semantics by means of PNs for the majority of the concepts of sequence diagrams. This semantics allows the concurrent behaviour of the diagrams to be modelled and subsequently analysed. Moreover, the usage of high-level PNs with data representation in its tokens permits an efficient structure for data types and control elements. In their approach, they use places to represent the messages, instead of transitions as we do. The authors consider the translation of UML 2.0 sequence diagrams to M-nets (Multivalued nets), a high level PNs with the usual inscriptions governing colored token flow and additional inscriptions governing

composition and synchronization of nets [Best et al. 1998]. They do not define the semantics of time events into the diagrams. They restrict parameters given to messages in constants attributes or variables of data type Boolean and Integer. Conditions are translated to transition guards, which assure that firing a transition is only possible if the respective condition is satisfied. They built a semantics of an interaction in a bottom-up way, starting from the innermost elements and incrementally adding surrounding elements on each level of nesting. To define the semantics they define maximal independent sets of partial lifelines, whose elements are completely unordered with respect to all other elements that are not part of the same lifeline. Thus, these lifelines fragments do not need special care for sequentialization. The semantics of each message independent set is defined compositionally and separately for each lifeline that is part of the set. All lifelines are put in parallel due to their independent behaviour. Send and receive events as well as data access inside conditions, actions, and message parameters are only handled by inscriptions of the PNs. The resulting nets have to be completed in a last step to gain explicit representations of PN semantics. A detailed definition of the transformation is presented, namely for both elementary diagram elements and combined fragments.

An aspect-oriented scenario modelling approach, to be used at the requirements level is presented in [Whittle and Araújo 2004]. Aspectual scenarios are modelled using interaction pattern specifications and composition of non aspectual requirements, which is represented by UML sequence diagrams. A technique for composing aspectual scenarios using an instantiation of an interaction pattern specification and an algorithm to synthesise state machines is introduced. The result is a set of state machines that represent the composed behaviour from the aspectual and nonaspectual scenarios. When executing these state machines, it is possible to simulate the combined aspectual and non aspectual behaviour at an early stage of the development process.

Haugen et al. [2005] proposes an approach, called STAIRS, aiming to provide a formal foundation for the use of UML interactions in a step-wise and incremental system development. STAIRS views the process of developing the interactions as a process of learning by describing. From a fuzzy and rough sketch, the aim is to reach a precise and detailed description applicable for formal handling. To come from the rough and fuzzy to the precise and detailed, STAIRS distinguishes among three main sub-activities:

- **supplementing:** categorizes (to this point) inconclusive behaviour as either positive or negative, recognizing that early descriptions normally

lack completeness. The initial requirements concentrate on the most obvious exceptional ones;

- **narrowing:** means reducing the allowed behaviour to better match the problem;
- **detailing:** involves introducing a more detailed description without significantly altering the externally observable behaviour.

The focus is on the refinement of sequence diagrams as a means for system development and validation.

Bontemps [2005] presents a way to create distributed reactive systems, based on modelling languages and their automated analysis. To narrow the gap between requirements and design models they suggest to follow two steps. Firstly, a complete behavioural specification of all user requirements (usually given by examples) is built. Secondly, the specification is transformed into the whole behaviour of each component of the system.

Hinchey et al. [2005] propose a round trip engineering approach, called R2D2C (Requirements to Design to Code), where designers write requirements as scenarios in constrained (domain specific) natural language. Other notations, including UML use cases, are however also possible. Based on the requirements, an equivalent formal model, using CSP, is derived, which is then used as a basis for code generation.

Uchitel and Kramer [2001] present a basis for a common approach to scenario-based specification, synthesis and analysis. Uchitel et al. [2003] present an MSC language with semantics in terms of labeled transition systems and parallel composition. The language integrates other languages based on the usage of high-level MSCs and on the identification of component states. With their language, scenario specifications can be broken up into manageable parts using high-level MSCs. These authors also present an algorithm that translates scenarios into a specification in the form of Finite Sequential Processes, which can be used for model checking and animation purposes. First, they define an MSC language with sound abstract semantics in terms of labeled transition systems and parallel composition. The language integrates existing approaches based on scenario composition by using high-level MSCs and those based on state identification by introducing explicit component labeling. This combination allows them to introduce additional domain-specific information and general assumptions explicitly into the scenario specification state labels. Second, they provide a synthesis algorithm which translates scenarios into a behavioural specification in the form of Finite Sequencial Processes [Magee and Kramer 1999]. This specification can

be analyzed with the Labeled Transition System Analyzer, which is a verification tool for concurrent systems, using model checking and animation. After that, they can demonstrate how many of the assumptions embedded in existing synthesis approaches can be made explicit and modeled in their approach.

The project P–UMLaut [2007] allows to develop tools to prevent inconsistencies, and expensive erroneous implementations of diagrams elements due the lacking of their precise semantic. The tools offered by P–UMLaut are used to design, simulation and prototyping of correct systems. The tools enhance system specifications providing a precise description of the diagrams. The simulation and animation components of the tools are used for the visualisation of the system behaviour. System executions can be analysed interactively in a 3D environment. P–UMLaut supports the following specification languages: UML 2.0 Sequence Diagrams, SDL, and BPEL. A formal description is automatically generated from the model, thereby exactly representing the system's behaviour by means of PNs. The adaptable simulator allows for execution of the modelled systems and interacts with various output targets via dedicated interfaces. In addition to the 3D animation, direct control of hardware and software systems is possible. P-UMLaut is thus an all-purpose engine for model-based control. Visualisation in P-UMLaut is accomplished by means of user defined 3D worlds. Already in the early phases of complex systems design, the model, its possible runs, and test cases can be demonstrated, simulated, visually analysed, and debugged. The formal semantics used in P-UMLaut enable automatic and interactive analysis of models. The verification component adds the possibility of a provable examination of system properties. Counterexamples found during this process can be simulated and hence point out flaws in the design. The p–UMLaut tool is divided into three main sections, which are the following:

- the description of the modelled system as an UML 2.0 sequence diagram;
- the Simulator which executes the PNs built by the Compiler;
- the 3D animation based on the IRRLICHT engine.

Firstly, the user creates different XML-files describing the sequence diagrams, 3D world and a xml-file mapping UML events and 3D animations or user events. According to the described system world the user has to create the 3D models.

Desharnais et al. [1998] present a way to represent scenarios as a relation between states (a state-oriented setting). This representation can be

graphical, due to relational transition systems. They make a distinction between environment moves and system moves, allowing moves "within" the environment and "within" the system, as we do it. A scenario is assumed to describe possible environment inputs and all legal system reactions. The authors propose an operator for integrating scenarios, based on the "demonic meet" operator. Like in our work, the integration of two scenarios relative to the same input obliges the system to answer as specified by both scenarios.

There are several work on the use of Petri nets to model the semantics of scenario-based models, namely MSCs [Heymer 2000; Kluge 2002; Rudolph et al. 1993].

Elkoutbi and Keller [1998] suggest the usage of use case diagrams and scenarios to obtain one hierarchical CPN model for the behaviour of an interactive system. The hierarchy of the CPN model mimics the one of the use case diagram. The usage of the colours in the nets preserves the independence of several scenarios after their integration in to the CPN model. This permits modelling of concurrency among use cases, scenarios and copies of the same scenario. However, their approach only tackles the controller perspective, and does not include the environment parts.

The work presented in [Elkoutbi and Keller 2000; Elkoutbi et al. 2006] includes two scenario-based techniques to generate user interfaces (UI). One technique suggests an approach for requirements engineering, linking UML models with UI prototypes. It provides a process involving five activities to derive a UI prototype from scenarios and to generate a formal specification of the application. Scenarios are acquired in the form of UML collaboration diagrams and are enriched with UI information. These diagrams are automatically transformed into the UML statechart specifications of all involved objects.

The other technique substitutes the use of statecharts by CPN models, that is, it automatically transforms the UML sequence diagrams into CPN models. It aims to model separately the use case and the scenario levels. The four activities leading from scenarios to executable UI prototypes are the following:

- Scenario Acquisition
- Specification Building
- Scenario Integration
- UI Prototype Generation

In the Scenario Acquisition, the analyst elaborates the use case diagram capturing the system functionalities, and for each use case, he or she acquires the corresponding scenarios in the form of sequence diagrams. Scenarios of a given use case are classified by type and ordered by frequency of use. They consider two types of scenarios: normal scenarios, which are executed in normal situations, and scenarios of exception executes in case of errors and abnormal situations. The frequency of use (or the frequency of execution) of a scenario is a number between 1 and 10 assigned by the analyst to indicate how often a given scenario is likely to occur.

The Specification Building consists on deriving CPN models from both the acquired used case diagram and all sequence diagrams. The CPN corresponding to the use case diagram is derived by mapping use cases into places. The transition leading to one place (*Enter*) corresponds to the initiating action of the use case. A place *Begin* may contain several tokens to model concurrent executions. In a use case diagram, a use case can call upon the services of another one via the relation *uses*, thus this must be specified at this level.

For each scenario of a given use case, one associates a table of object states, which is directly obtained from the sequence diagram of the scenario by following the exchange of messages from top to bottom and identifying the changes in object states caused by the messages. From each object state table, a CPN model is generated by transforming scenario states into places, and messages into transitions. Each scenario is assigned a distinct colour. When specifying scenarios, only the object state tables is manually obtained. The rest of the operation is fully automatic.

In the next activity of the approach, Scenario Integration, all CPN models corresponding to the scenarios of a use case are merged, in order to produce an integrated CPN model for the behaviour of the use case. Given two CPN models that correspond to two different scenarios the algorithm proposed in [Elkoutbi and Keller 1998] merges all places in the CPN models with the same names. The merged places will have as color the union of the colors of the two scenarios. Then, the algorithm looks for transitions having the same input and output places in the two scenarios and merges them with an *OR* of their guard conditions.

When integrating several scenarios, the resulting specification captures in general not only the input scenarios, but perhaps even more, they call it the interleaving problem [Elkoutbi and Keller 1998], which is solved by introducing the so called chameleon token (a token that can take on several colours). An integrated CPN model corresponding to a given use case can be connected to the CPN model derived from the use case diagram.

In the user UI Prototype Generation, a UI prototype of the system is derived from the CPN model. The generated prototype is standalone and comprises a menu to switch between the different use cases. The various screens of the prototype represent the static aspect of the UI; the dynamic aspect of the UI, as captured in the CPN models, map into the dialog control of the prototype. The activity of prototype generation includes of the following five operations:

- Generating graph of transitions
- Masking non-interactive transitions
- Identifying UI blocks
- Composing UI blocks
- Generating frames from composed UI blocks

Küster-Filipe [2006] is gives a semantics of sequence diagrams, combined with an OCL liveness, using labelled event structures [Winskel and Nielsen 1995], and shows how sequence diagrams can be embedded into a true-concurrent two-level logic interpreted over labelled event structures [Küster-Filipe 2000]. The top level logic, called communication logic, is used to describe inter-object specification, that is it describes interactions among several instances. The lower level logic, called home logic, describes intra-object behaviour, such as state invariants and interactions constraints. Thus, Küster-Filipe's work presents a concurrent distributed temporal logic and shows how interactions and various constraints can be described in this logic. The work indicates two ways to use that logic. Firstly, to capture some interaction properties to check whether the inter-object behavioural model (a labelled event structure) satisfies the properties. Secondly, to capture the entire interaction of a sequence diagram as a set of formulae.

Kloul and Küster-Filipe [2005] show a way of modelling mobility and performance information at design level using the interaction overview diagrams (IODs), and UML2 sequence diagrams, the performance modelling technique PEPA (Perfomance Evaluation Process Algebra) nets [Gilmore et al. 2003], the result of the combination of coloured stochastic PNs and the stochastic process algebra formalism also called PEPA. They describe the formal translation of IODs into PEPA nets. This translation allows designers using UML2.0, to formally analyse their models formally using the PEPA workbench for PEPA nets. The mobility is modeled in UML2.0 notation in the following way:

- the locations and movements of objects between locations are described in the high level of IOD;
- the locally behaviour and interaction of objects is described in the low level of IOD by the individual nodes of the IOD, which are sequence diagrams.

Both levels of IODs are enriched with performance related information, by indicating, at the source pin of an IOD edge, the explicit action that corresponds to the movement of an object from one location to another. With this information, a PEPA net model can be derived. There is a direct correspondence between the IOD nodes and the objects in the UML model, with the places and the components in the PEPA net model. The idea of modelling mobility, with UML notation, for the performance analysis is not new [Grassi et al. 2004].

Giese et al. [2005] present an approach to integrate scenario-based models with the state-based models in the design and synthesis of a controller for flexible production systems, in order to obtain the expected behaviour. Scenarios describe the expected behaviour of the controller and the work shows how the scenarios can be used to record observed behaviour for analysis or 3D-visualization.

The work presented in thesis work is based on the results presented in [Machado et al. 2005], where the authors show how the behaviour of animation prototypes results from the translation of sequence diagrams into CPN models. We extend their results by showing how to translate more types of operators in UML 2.0 sequence diagrams, namely by considering parallel constructors which result in CPN models with true concurrency (i.e., CPN models that are not just sequential machines).

7.4 Animating Formal Specifications of Requirements

In some works, [Morrey et al. 1998; Vázquez 2001] the animation is used to mean a creation of a prototype following a formal specification.

Several works suggest the usage different visualizations of requirements specifications. The visual representations provide cognitive support by highlighting the most relevant aspects of a specification and interactions.

Dulac et al. [2002] propose a set of principles and a taxonomy for creating visual representations corresponding to requirements specifications described in a formal language. The principles and taxonomy are illustrated

by sample visualizations created while understanding a formal specification of a flight management system.

Other works suggest the usage of graphic elements to animate the requirements. Burd et al. [2002] suggest that sequence diagrams can be better understood through the usage of animation to evaluate the control flow described in the sequence diagram.

Van et al. [2004] present a tool for the animation of goal-oriented requirements. The model is obtained using model synthesis techniques, and supports animation and validation of property-based specifications. Ponsard et al. [2005] also present an approach to support animation and validation of property-based specifications.

Magalhães et al. [1998] introduce the use of PNs and some of their extensions for animation. The authors argue that the usage of PNs can improve the techniques for modelling different aspects of the behaviour of an animation.

Holzmann et al. [1997] present a set of three tools to support the formalization of requirements, to organize, and to make queries in the obtained formalization. These authors consider MSCs as the formal language to be used. The first tool is based on a graphical environment to create and edit MSCs, that allows also the association of informal text descriptions to the elements in the diagram. The second tool, called POGA, is used to organize the existing MSCs creating a graph containing the interdependencies among them, and generating visual display of the obtained graph. The third tool is the TEMPLE to search fragments of MSCs along the graph representing the interdependencies of MSCs. Notice that the POGA tool aims to display the interdependencies between the MSCs, but not the behaviour present in each MSCs in a different notation, as we do in this work for the sequence diagrams. These tools facilitate the manipulation of the obtained formal specifications of requirements, which is different from our goal of offering an animation view of the problem domain.

Gargantini et al. [2009] presents a method to validate embedded system designs provided in terms of UML models, by transforming the considered UML models into abstract state machines (ASM) formal models, and allowing the simulation-based validation of formal models.

Ermel and Bardohl [2004] present an approach to extend formal specifications with specifications of graphical animations using a graph-grammar approach. This extension was done over the GenGED tool [Bardohl 2002], that proposes the automatic generation of a visual environment to manipulate visual models based on a specified visual language. This approach is illustrated in [Ehrig et al. 2006], where the PN modelling language is consid-

ered as an example of the formal language to be used on the formalization of an elevator example. The approach to create an animation view is not focused in a specific modelling language, as does our approach with the CPN modelling language. Another difference is that they consider the modelling using the language to be animated, and we consider that the CPN to be animated are obtained from scenario descriptions by sequence diagrams. The authors consider a notion of animation, very close to ours in the sense that it is not considers a prototype generated from a formal specification, but an animation of the problem domain.

Chapter 8

Conclusions and Future Work

Summary

This chapter presents a summary of the contributions resulting from the work described in this dissertation document, and points out several ideas for future work.

Contents

8.1 Contributions 170
8.2 Future Work 171

8.1 Contributions

The contribution of this thesis is an approach to validate, through animation, a reactive system that is described by a set of behavioural scenarios. The method transforms these scenarios into a state-based model that also includes the behaviour of elements from the environment. Tool support for the transformation is partially supported in order to illustrate that the transformation can be automated with state-of-the-art technologies.

In particular, this work is focused in obtaining a CPN model, i.e., our approach allows a CPN model to be generated from a set of scenarios, expressed as UML2 sequence diagrams. The natural support to parallelism and concurrency given by the CPN modelling language permits the CPN model to be considered for simultaneous execution of several scenarios (either of the same use case or of different use cases) and also to represent the parallel activities inside a scenario. Therefore, the CPN models support two types of parallelism (intra-scenario and inter-scenario), which means that they can be made rich enough to explore situations where several scenarios that might affect each other are executed simultaneously.

The method also supports the modelling of some aspects of the human users, namely some possible unexpected behaviours. This feature is very important to address more cases in the animation, and thus better validate the requirements with the stakeholders.

Our approach easily scales up, since a CPN model distributes the system's complexity among its graphical and textual parts. For example, in the system for the check-in of passengers (see Section 6.3), the introduction of more passengers and check-in agents does not change the structure of the CPN model; we only need to add extra tokens to some well identified places. Additionally, we propose scenarios to be reflected in the CPN model in an iterative way. This implies that when, for example, the first alternative scenario is being considered, we should update the CPN model obtained for the main scenario with the extra behaviour introduced by the alternative or exception scenarios. Typically, one expects the extra features to be much smaller than all behaviour, since there are some overlapping parts in the scenarios (of the same use case).

During this thesis work some efforts have been conducted to study the feasibility of accomplishing the transformation of a set of scenarios into a CPN model in an automatic way. A tool was developed to support the transformation of a set of sequence diagrams into a unique CPN model. This tool must be understood as a proof-of-concept to illustrate that this transformation can be automated with state-of-art paradigms and technologies.

For validation purposes, our approach suggests the development of a graphical animation layer over the obtained state-based model. This thesis suggests some guidelines to enrich the CPN model with some animation-specific code. Due to the support for parallelism and concurrency given by a CPN model, the behaviours in the animation can be executed simultaneously, which allows one to detect some situations where scenarios or use cases affect each other.

The animation layer consists on a set of graphical elements that are intended to represent the elements in the context of the system being developed. When creating an animation, the software engineer must base her work on the requirements document information, and on her perception about what turns an animation easy to understand for the stakeholders. During this activity some changes may be introduced into the CPN model in order to obtain the expected animations when executing the CPN model together with the animation specification. The BRITNeY suite animation tool is used to connect in the CPN tools the execution of the CPN model with the animation, namely by using the SceneBeans plug-in. Our work provides some guidelines on how to introduce in the CPN model the code related to the animation.

The animations used in this work are based on SceneBeans objects that are specified in XML-based file. To facilitate the generation and manipulation of the XML-files used to define SceneBeans animation, we propose a set of scripts to enclose and to hide some of the XML syntax details.

In general terms, this work is a contribution to the combination of the UML and Petri nets, which is considered to be useful to the software engineering area by some authors [Denaro and Pezzè 2004; Gomaa 2006].

8.2 Future Work

As future work, we plan to apply our approach in industrial contexts in order to evaluate how useful and practical it proves to be for software practitioners. To facilitate the practical application of the approach, we expect to improve the tool support for the automatic transformation from scenarios into CPN models.

Our approach that for describing requirements adopts scenarios as the main source model can be improved and made more effective if complemented with other techniques. The adoption of those techniques must be carefully evaluated, since they should truly complement scenarios, and not address the same perspectives. For example, goal modelling approaches and

scenario-based techniques can be complementary used [van Lamsweerde and Willemet 1998]. Similarly, it seems that personas, goals and scenarios constitute also a useful combination for requirements engineering [Aoyama 2007].

There are some other usages for the obtained CPN model, namely the exploration of the verification of some properties using the available techniques like model checking for that purpose.

An interesting point to be studied in the future is the usage of the obtained CPN model along to support the rest of the development activities on the development process, such as the obtainment of a first architecture for the software system under consideration.

We plan also to extend our work to support some different kinds of graphical animations, that can be more adequate for some specific situations, such as the ones where there is a continuous behaviour to be animated. There is an ongoing project on the usage of virtual worlds of *Second Life* as animations (in three dimensions), whose avatars represent the human users of the system under development. This context is specially interesting and useful for analysing the user experience within a complex software-based system.

Bibliography

A. Abran, J. W. Moore, P. Bourque, R. Dupuis, and L. L. Tripp. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE, 2004. URL `http://www.swebok.org/`. 39, 40

M. A. Adamski. Direct Implementation of Petri Net Specification. In *7th International Conference on Control Systems and Computer Science*, pages 74–85, 1987. 94, 106

M. A. Adamski, A. Karatkevich, and M. Wegrzyn, editors. *Design of Embedded Control Systems*, 2005. Springer. 26

L. Amorim, P. Maciel, M. Nogueira, R. Barreto, and E. Tavares. A Methodology for Mapping Live Sequence Chart to Coloured Petri Net. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 2999–3004, October 2005. DOI `10.1145/1127878.1127880`. 159

M. Andersson and J. Bergstand. Formalizing Use Cases with Message Sequence Charts. Master's thesis, Lund Institute of Technology, 1995. 156

M. Aoyama. Persona-Scenario-Goal Methodology for User-Centered Requirements Engineering. In *15th IEEE International Requirements Engineering Conference (RE 2007)*. IEEE CS Press, 2007. 172

A. Aurum and C. Wohlin. *Engineering and Managing Software Requirements*. Springer Berlin Heidelberg, 2005. DOI `10.1007/3-540-28244-0`. 38, 187

I. J. Ball and T. C. Ormerod. Putting Ethnography to Work: the Case for a Cognitive Ethnography of Design. *International Journal of Human-Computer Studies*, 53(1):147–68, July 2000. 48

R. Bardohl. A Visual Environment for Visual Languages. *Science of Computer Programming*, 44(2):181–203, 2002. ISSN 0167-6423. DOI `10.1016/S0167-6423(02)00038-2`. 167

E. Best, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz. M-nets: An Algebra of High-level Petri Nets, with an Application to the Semantics of Concurrent Programming Languages. *Acta Informatica*, 35(10):813–857, 1998. DOI 10.1007/s002360050144. 160

J. Billington, M. Diaz, and G. Rozenberg, editors. *Application of Petri Nets to Communication Networks, Advances in Petri Nets*, London, UK, 1999. Springer-Verlag. ISBN 3-540-65870-X. 26

J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, pages 49–70, 2004. 26

R. M. Blanco. Requirements Specification for an Elevator Controller. Technical report, School of Computer Science, University of Waterloo, Canada, 2005. URL http://www.student.cs.uwaterloo.ca/~cs445/Winter2006/Project/SRS-Rolando-Blanco.pdf. 125

B. W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, January 1984. DOI 10.1109/MS.1984.233702. 6

B. W. Boehm. A spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988. DOI 10.1109/2.59. 42

B. W. Boehm, P. Grünbacher, and R. O. Briggs. Developing Groupware for Requirements Negotiation: Lessons Learned. *IEEE Software*, 18(3): 46–55, 2001. ISSN 0740-7459. DOI 10.1109/52.922725. 43

Y. Bontemps. *Relating Inter-Agent and Intra-Agent Specifications (The Case of Live Sequence Charts)*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), April 2005. 161

J. A. Bubenko and B. Wangler. Objectives Driven Capture of Business Rules and of Information Systems Requirements. *International Conference on Systems, Man and Cybernetics (ICSMC 1993)*, 1:670–7, October 1993. DOI 10.1109/ICSMC.1993.384821. 49

E. Burd, D. Overy, and A. Wheetman. Evaluating Using Animation to Improve Understanding of Sequence Diagrams. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, Missouri, USA, 2002. 167

J. Bézivin, F. Jouault, and P. Valduriez. An Eclipse-Based IDE for the ATL Model Transformation Language. Technical Report 04.08, LINA Research, , University of Nantes, 2004. 67

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000. 107

CPN Tools, 2009. URL `http://www.daimi.au.dk/CPNtools`. 9, 35, 70

K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–40, 2006. DOI `10.1147/sj.453.0621`. 67

B. Dano, H. Briand, and F. Barbier. An Approach Based on the Concept of Use Case to Produce Dynamic Object-Oriented Specifications. In *3rd IEEE International Symposium on Requirements Engineering (RE 1997)*, pages 54–64. IEEE CS Press, 1997. ISBN 0-8186-7740-6. DOI `10.1109/ISRE.1997.566842`. 8, 156

A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed Requirements Acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993. ISSN 0167-6423. DOI `10.1016/0167-6423(93)90021-G`. 49

G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994. 107

G. Denaro and M. Pezzè. Petri Nets and Software Engineering. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 439–466. Springer, January 2004. DOI `10.1007/b98282`. 6, 171

J. Desharnais, M. Frappier, R. Khédri, and A. Mili. Integration of Sequential Scenarios. *IEEE Transactions on Software Engineering*, 24(9):695–708, September 1998. DOI `10.1109/32.713325`. 162

A. A. Desrochers and R. Y. Al-Jaar. *Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis*. IEEE, 1994. 26

N. Dulac, T. Viguier, N. Leveson, and M.-A. Storey. On the Use of Visualization in Formal Requirements Specification. In *Proceedings of IEEE Joint International Conference on Requirements Engineering*, pages 71–80, 2002. 166

Eclipse UML2. Eclipse UML2 project web page. online, 2008. URL `http://www.eclipse.org/uml2`. 68

H. Ehrig, C. Ermel, and G. Taentzer. Simulation and Animation of Visual Models of Embedded Systems: A Graph-Transformation-Based Approach Applied to Petri Nets. In G. Hommel and H. Sheng, editors, *Proceedings of 7th Workshop on Embedded Systems – Modeling, Technology, and Applications, Technische Universität Berlin*, pages 11–20. Springer Verlag, 2006. DOI `10.1007/1-4020-4933-1`. 167

C. Eichner, H. Fleischhack, R. Meyer, U. Schrimpf, and C. Stehno. Compositional Semantics for UML 2.0 Sequence Diagrams Using Petri Nets. In *12th International SDL Forum (SDL 2005)*, volume 3530 of *Lecture Notes in Computer Science*, pages 133–148, Grimstad, Norway, January 2005. Springer. 159

M. Elkoutbi and R. K. Keller. Modeling Interactive Systems with Hierarchical Colored Petri Nets. In *Advanced Simulation Technologies Conference (ASTC 1998)*, pages 432–37, 1998. 8, 163, 164

M. Elkoutbi and R. K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In M. Nielsen and D. Simpson, editors, *21st International Conference on Application and Theory of Petri Nets (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 166–186, Aarhus, Denmark, June 2000. Springer. 163

M. Elkoutbi, I. Khriss, and R. K. Keller. Automated Prototyping of User Interfaces based on UML Scenarios. *Automated Software Engineering, Kluwer Academic Publishers*, 1:5–40, 2006. DOI `10.1007/s10515-006-5465-5`. 163

C. Ermel and R. Bardohl. Scenario Animation for Visual Behavior Models: A Generic Approach. *Software and System Modeling*, 3(2):164–177, 2004. DOI `10.1007/s10270-003-0048-4`. 167

J. M. Fernandes, A. M. Pina, and A. J. Proença. Concurrent Execution of Petri Nets based on Agents. In *Encontro Nacional do Colégio de Engenharia Electrotécnica (ENCEE 1995)*, pages 83–89, Lisbon, Portugal, December 1995. Ordem dos Engenheiros. 94, 95

J. M. Fernandes, M. A. Adamski, and A. J. Proença. VHDL Generation from Hierarchical Petri Net Specifications of Parallel Controller. *IEE*

Proceedings: Computers and Digital Techniques, 144(2):127–37, March 1997. 106, 107

W. Foddy. *Constructing Questions for Interviews and Questionnaires*. Cambridge University Press, Cambridge, 1993. ISBN 0-521-46733-0. 46

M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modelling Language*. Addisson-Wesley, 2004. 7

F. Fransella. Some Skills and Tools for Personal Construct Practitioners. In *International Handbook of Personal Construct Psychology*, pages 105–121. John Wiley & Sons, Ltd, 2004. DOI 10.1002/0470013370.ch10. 47

A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven Design and ASM-based Validation of Embedded Systems. In L. Gomes and J. M. Fernandes, editors, *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*, pages 24–55. Information Science Reference, July 2009. 167

H. Giese, E. Kindler, F. Klein, and R. Wagner. Reconciling Scenario-centered Controller Design with State-based System Models. In *4th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2005)*, pages 1–5, NY, USA, 2005. ACM Press. ISBN 1-58113-963-2. DOI 10.1145/1083183.1083187. 166

S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: A Structured Performance Modelling Formalism. *Performance Evaluation*, 54(2):79–104, 2003. ISSN 0166-5316. DOI 10.1016/S0166-5316(03)00069-5. 165

M. Glinz. An integrated formal model of scenarios based on statecharts. In W. Schäfer and P. Botella, editors, *5th European Software Engineering Conference*, pages 254–271. Springer, 1995. 156

J. A. Goguen and C. Linde. Techniques for Requirements Elicitation. In *International Symposium on Requirements Engineering (RE 1993)*, pages 152–164, Los Alamitos, California, 1993. IEEE CS Press. 46, 47, 48

H. Gomaa. A Software Modeling Odyssey: Designing Evolutionary Architecture-Centric Real-Time Systems and Product Lines. In *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, Lecture Notes in Computer Science, pages 1–15, Genova, Italy, 2006. Springer. DOI 10.1007/11880240_1. 171

H. Gomaa and D. B. Scott. Prototyping as a Tool in the Specification of User Requirements. In *5th International Conference on Software Engineering (ICSE 1981)*, pages 333–342, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. 48

E. Gottesdiener. *Requirements by Collaboration: Workshops for Defining Needs*. Addison-Wesley Professional, April 2002. ISBN 0201786060. 48

V. Grassi, R. Mirandola, and A. Sabetta. UML based modeling and performance analysis of mobile systems. In *7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2004)*, pages 95–104, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-953-5. DOI 10.1145/1023663.1023683. 166

D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, January 2001. (Also, Proc. Fundamental Approaches to Software Engineering (FASE; invited paper), Lecture Notes in Computer Science, Vol. (Tom Maibaum, ed.), Springer-Verlag, March 2000, pp. 22-34.). 157

D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: the Play-in/Play-out Approach. *Software and System Modeling*, 2 (2):82–107, 2003a. DOI 10.1007/s10270-002-0015-5. 157

D. Harel and R. Marelly. *Come, Let's Play! Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003b. ISBN: 3-540-00787-3. 157

D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-out. In R. Crocker and G. L. Steele Jr., editors, *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 68–69, October 26-30 2003. DOI 10.1145/949344.949353. 158

O. Haugen, K. E. Husa, R. K. Runde, and K. Stolen. STAIRS Towards Formal Design with Sequence Diagrams. *Software and Systems Modeling*, 2:1–13, June 2005. 160

P. Haumer, K. Pohl, and K. Weidenhaupt. Requirements Elicitation and Validation with Real World Scenes. *IEEE Transactions on Software Engineering*, 24(12):1036–54, Dec 1998. ISSN 0098-5589. DOI 10.1109/32.738338. 49

S. Heymer. A Semantics for MSCs based on Petri Nets Components. In *Second Conference on SDL and MSC (SAM 2000)*, Grenoble, June 2000. 163

M. G. Hinchey, J. L. Rash, and C. A. Rouff. A Formal Approach to Requirements-Based Programming. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2005)*, pages 339–45. IEEE CS Press, 2005. ISBN 0-7695-2308-0. DOI 10.1109/ECBS.2005.7. 161

K. Holtzblatt and H. R. Beyer. Requirements Gathering: the Human Factor. *Communications of the ACM*, 38(5):31–32, 1995. ISSN 0001-0782. DOI 10.1145/203356.203361. 46

G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991. ISBN 0-13-539925-4. 107

G. J. Holzmann. *The Spin Model Checker: Primier and Reference Manual*. Addison-Wesley, September 2003. 107

G. J. Holzmann, D. A. Peled, and M. H. Redberg. Design Tools for Requirements Engineering. *Bell Labs Technical Journal*, 1:86–95, Winter 1997. 167

P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, and C. Chen. Formal Approach to Scenario Analysis. *IEEE Software*, 11(2):33–41, 1994. ISSN 0740-7459. DOI 10.1109/52.268953. 156

M. Jackson. *Problem Frames: Analysing & Structuring Software Development Problems*. Addison-Wesley, 2000. 47, 147

D. Jankowicz. *The Easy Guide to Repertory Grids*. Wiley, 2003. 47

K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Berlin Heidelberg, 2009. DOI 10.1007/b95112. 26, 29

K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Journal on Software Tools for Technology Transfer*, 9(3-4):213–54, June 2007. DOI 10.1007/s10009-007-0038-x. 9, 35

M. Jirotka and J. A. Goguen. *Requirements Engineering: Social and Technical Issues*. Academic Press Professional, Inc., San Diego, CA, USA, 1994. ISBN 0-12-385335-4. 48

J. B. Jørgensen and K. B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS 2006)*, pages 259–268, Washington, DC, USA, 2006. IEEE Computer Society. 35

F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. ISSN 0167-6423. DOI 10.1016/j.scico.2007.08.002. 67

G. Juhás, R. Lorenz, and J. Desel. Can I Execute My Scenario in Your Net? In G. Ciardo and P. Darondeau, editors, *26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 381–390, Miami, USA, June 2005. Springer. DOI 10.1007/b136988. 159

L. D. Kaufman, S. M. Thebaut, and M. F. Intemnte. System Modeling for Scenario-Based Requirements Engineering. Technical report, SERC, USA, 1989. 50

W. D. Kelton, R. P. Sadowski, and D. T. Sturrock. *Simulation with Arena*. McGraw-Hill, third edition edition, 2004. 54

J. Kleijn and M. Koutny. Process Semantics of P/T-Nets with Inhibitor Arcs. In *ICATPN*, pages 261–81, 2000. 109

L. Kloul and J. Küster-Filipe. From Interaction Overview Diagrams to PEPA Nets. In *4th Workshop on Process Algebras and Timed Activities (PASTA 2005)*, Edinburgh, UK, 2005. 165

O. Kluge. *Compositional Semantics for Message Sequence Charts based on Petri Nets*. PhD thesis, Technische Universität Berlin, Germany, September 2002. 163

L. Kristensen, M. Westergaard, and P. Nørgaard. Model-Based Prototyping of an Interoperability Protocol for Mobile Ad-Hoc Networks. In *5th International Conference on Integrated Formal Methods (IFM 2005)*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, Oct. 2005. DOI 10.1007/11589976_16. 35

I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999. 158

J. Küster-Filipe. Fundamentals of a Module Logic for Distributed Object Systems. *Journal of Functional and Logic Programming*, 3:5–10, March 2000. 165

J. Küster-Filipe. Modelling Concurrent Interactions. *Theoretical Computer Science, Algebraic Methodology and Software Technology*, 351(2):203–220, February 2006. 165

A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, third edition edition, 2000. 51, 53

W. J. Lee, S. D. Cha, and Y. R. Kwon. Integration and Analysis of Use Cases using Modular Petri Nets in Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(12):1115–30, December 1998. 157

S. Leue and P. B. Ladkin. Implementing and Verifying Scenario-Based Specifications Using Promela/XSpin, 1996. 157

F. Lustman. A Formal Approach to Scenario Integration. *Ann. Softw. Eng.*, 3:255–71, 1997. ISSN 1022-7091. 157

R. J. Machado, J. M. Fernandes, and A. J. Proença. Specification of Industrial Digital Controllers with Object-Oriented Petri Nets. In *IEEE International Symposium on Industrial Electronics (ISIE 1997)*, volume 1, pages 78–83, July 1997. DOI `10.1109/ISIE.1997.651794`. 94, 95

R. J. Machado, K. B. Lassen, S. Oliveira, M. Couto, and P. Pinto. Execution of UML Models with CPN Tools for Workflow Requirements Validation. In *Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, 2005. 35, 166

L. P. Magalhães, A. B. Raposo, and I. L. M. Ricarte. Animation Modeling with Petri Nets. *Computers & Graphics*, 22(6):735–43, 1998. 167

J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999. 161

J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behavior Models. In *International Conference on Software Engineering (ICSE 2000)*, pages 499–508, Limerick, Ireland, 2000. 9, 87

N. Maiden. CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements. *Automated Software Engineering*, 5(4):419–446, 1998. DOI `10.1023/A:1008605412971`. 50

Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, USA, 1992. 4, 107

I. Morrey, J. Siddiqi, R. Hibberd, and G. Buckberry. A Toolset to Support the Construction and Animation of Formal Specifications. *Journal of Systems Software*, 41(3):147–160, 1998. ISSN 0164-1212. DOI 10.1016/S0164-1212(97)10016-4. 166

M. Mukund. Petri Nets and Step Transition Systems. *International Journal of Foundations of Computer Science*, 3(4):443–78, 1992. 112

T. Murata. Petri Nets: Properties, Analysis and Applications. In *Procedings of the IEEE*, pages 541–80, April 1989. 107, 113

B. Nuseibeh and S. Easterbrook. Requirements Engineering: a Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (FOSE 2000)*, pages 35–46, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-253-0. DOI 10.1145/336512.336523. 45

B. Nuseibeh, A. Finkelstein, and J. Kramer. Method Engineering for Multi-perspective Software Development. *Information and Software Technology*, 38(4):267–72, 1996. 50

Object Management Group. Unified Modeling Language: Superstructure Specification, version 2.1.2, August 2007. URL http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/. 16, 20, 21, 22, 23, 24, 25

P–UMLaut. Project P-UMLaut. Online, 2007. http://www.p-umlaut.de. 162

J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. ISBN 0136619835. 109

C. Ponsard, P. Massonet, A. Rifaut, J. F. Molderez, A. v. Lamsweerde, and H. T. Van. Early Verification and Validation of Mission-Critical Systems. *Electronic Notes in Theoretical Computer Science*, 133:237 – 254, 2005. DOI 10.1016/j.entcs.2004.08.067. Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2004). 167

C. Potts, K. Takahashi, and A. I. Anton. Inquiry-Based Requirements Analysis. *IEEE Software*, 11(2):21–32, 1994. ISSN 0740-7459. DOI 10.1109/52.268952. 49, 50

N. Pryce and J. Magee. SceneBeans: A Component-Based Animation Framework for Java. Online, 2007. URL `http://www-dse.doc.ic.ac.uk/Software/SceneBeans/`. 35, 87

W. Reisig. *Petri Nets - An Introduction*. Springer, Heidelberg, Germany, EATCS monographs on theorical computer science edition, 1985. 25, 31, 109, 112

W. Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer, 1998. 26

REXML. Ruby REXML Lybrary. Online, 2007. URL `www.germane-software.com/software/rexml/`. 90

Ó. R. Ribeiro and J. M. Fernandes. Some Rules to Transform Sequence Diagrams into Coloured Petri Nets. In *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006)*, pages 227–41, October 2006. URL `http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/`. 60

Ó. R. Ribeiro and J. M. Fernandes. On the Use of Coloured Petri Nets for Visual Animation. In *8th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2007)*, pages 223–241, October 2007a. URL `http://www.daimi.au.dk/CPnets/workshop07/cpn/papers/`. 78

Ó. R. Ribeiro and J. M. Fernandes. Validation of Reactive Software from Scenario-based Models. In *In Second Software Engineering Doctoral Consortium (SEDES 2007) at QUATIC 2007*, pages 213–7, Lisbon, Portugal, September 2007b. IEEE Computer Society Press. DOI `10.1109/QUATIC.2007.33`. 9

Ó. R. Ribeiro and J. M. Fernandes. Translating Synchronous Petri Nets into PROMELA for Verification of Behavioural Properties. In *Second IEEE International Symposium on Industrial Embedded Systems (SIES 2007)*, pages 266–73, Lisbon, Portugal, July 2007c. IEEE Computer Society Press. DOI `10.1109/SIES.2007.4297344`. 106

Ó. R. Ribeiro and J. M. Fernandes. Validation of Scenario-based Business Requirements with Coloured Petri Nets. In *The Fourth International Conference on Software Engineering Advances (ICSEA 2009)*, pages 250–255. IEEE Computer Society Press, September 2009. DOI `10.1109/ICSEA.2009.45`. 141

Ó. R. Ribeiro, J. M. Fernandes, and L. F. Pinto. Model Checking Embedded Systems with PROMELA. In *12th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2005)*, pages 378–85, Greenbelt, MD, USA, Apr. 2005. IEEE Computer Society Press. DOI 10.1109/ECBS.2005.53. 106, 108

J. Richardson, T. C. Ormerod, and A. Shepherd. The Role of Task Analysis in Capturing Requirements for Interface Design. *Interacting with Computers*, 9(4):367–384, 1998. 46

S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison Wesley Professional, second edition, 2006. 41, 48, 141, 142

C. Rolland, C. Souveyet, and C. B. Achour. Guiding Goal Modeling Using Scenarios. *IEEE Transactions on Software Engineering*, 24(12):1055–71, Dec 1998. ISSN 0098-5589. DOI 10.1109/32.738339. 49

Ruby. Ruby Programming Language. Online, 2007. URL http://www.ruby-lang.org. 90

E. Rudolph, J. Grabowski, and P. Graubmann. Towards a Petri Net Based Semantics Definition for Message Sequence Charts. In O. Faergemand and A. Sarma, editors, *SDL 1993 - Using Objects*, pages 193–208, October 1993. 163

Scenario Plus, 2008. URL http://www.scenarioplus.org.uk/. Accessed 29 July 2008. 50

S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, Sept-Oct 2003. DOI 10.1109/MS.2003.1231150. 67

C. Sibertin-Blanc, N. Hameurlain, and O. Tahir. Ambiguity and Structural Properties of Basic Sequence Diagrams. *Innovations in Systems and Software Engineering*, 4(3):275–284, 2008. DOI 10.1007/s11334-008-0063-2. 68

M. d. S. Soares and J. Vrancken. A Metamodeling Approach to Transform UML 2.0 Sequence Diagrams to Petri Nets. In *Proceedings of the IASTED International Conference Software Engineering*, Innsbruck, Austria, february 2008. 158

S. Some, R. Dssouli, and J. Vaucher. Toward an Automation of Requirement Engineering using Scenarios. *Journal of Computing and Information*, 2 (1):110–132, 1996. 157

I. Sommerville. *Software Engineering*. International Computer Sciences Series. Addison-Wesley, 8th edition, 2006. 38, 39, 41, 42, 44, 49, 51, 53

I. Sommerville, P. Sawyer, and S. Viller. Viewpoints for Requirements Elicitation: A Practical Approach. *ICRE*, 00:0074, 1998. ISSN 1097-0592. DOI 10.1109/ICRE.1998.667811. 50

Standard ML. Online:, 2006. http://www.smlnj.org. 27

H. Störrle. A Petri-net Semantics for Sequence Diagrams. In *GI/ITG Fachgespräch Formale Beschreibungstechniken für verteilte Systeme (FBT 1999)*, 1999. 159

A. Sutcliffe and N. Maiden. The Domain Theory for Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, Mar. 1998. DOI 10.1109/32.667878. 47

D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, second edition, October 2004. ISBN 0974514055. 90

S. Uchitel and J. Kramer. A Workbench for Synthesising Behaviour Models from Scenarios. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 188–197. IEEE Computer Society, 2001. 161

S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, Feb. 2003. DOI 10.1109/TSE.2003.1178048. 161

S. Uchitel, M. Broy, I. H. Krüger, and J. Whittle. Guest Editorial: Special Section on Interaction and State-Based Modeling. *IEEE Transactions on Software Engineering*, 31(12):997–998, 2005. DOI 10.1109/TSE.2005.139. 7

J. D. Ullman. *Elements of ML programming*. Prentice-Hall, 1998. 27

H. T. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. Goal-Oriented Requirements Animation. In *Proceedings of the 12th IEEE International*

Requirements Engineering Conference (RE 2004), pages 218–28, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2174-6. DOI 10.1109/RE.2004.24. 167

W. van der Aalst and K. van Hee. *Workflow Management : Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press, March 2004. 26

A. van Lamsweerde and L. Willemet. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–114, 1998. ISSN 0098-5589. DOI 10.1109/32.738341. 172

A. J. G. Vázquez. *Computer-Aided Validation of Formal Conceptual Models*. PhD thesis, Technical University of Braunschweig, 2001. 166

M. Westergaard. The BRITNeY Suite: A Platform for Experiments. In *7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006)*, October 2006. 78

M. Westergaard and K. B. Lassen. Building and Deploying Visualizations of Coloured Petri Net Models Using BRITNeY Animation and CPN Tools. In *Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, 2005. 105

M. Westergaard and K. B. Lassen. The BRITNeY Suite Animation Tool. In *27th International Conference on Applications and Theory of Petri Nets*, pages 431–440, 2006. DOI 10.1007/11767589_26. 9, 35, 78

J. Whittle and J. Araújo. Scenario Modelling with Aspects. *IEE Proceedings - Software*, 151(4):157–171, 2004. 160

J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *22nd International Conference on Software Engineering (ICSE 2000)*, pages 314–323, Limerick, Ireland, June 2000. 158

J. Whittle, J. Saboo, and R. Kwan. From Scenarios to Code: An Air Traffic Control Case Study. In *25th International Conference on Software Engineering(ICSE 2003)*, pages 490–497, 2003. 158

J. Whittle, R. Kwan, and J. Saboo. From Scenarios to Code: An Air Traffic Control Case Study. *Software and Systems Modeling*, 4(1):71 – 93, Feb 2005. 158

K. E. Wiegers. *Software Requirements*. Microsoft Press, second edition edition, 2003. 44

R. J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003. 4, 5

G. Winskel and M. Nielsen. Models for Concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1995. 165

J. Wood and D. Silver. *Joint Application Development*. Wiley, 2nd edition edition, 1995. 48

A. Yakovlev, L. Gomes, and L. Lavagno, editors. *Hardware Design and Petri Nets*, 2000. Springer. 26

E. S. K. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. *Requirements Engineering, IEEE International Conference on*, 0:226, 1997. ISSN 1090-705X. DOI 10.1109/ISRE.1997.566873. 49

P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997. ISSN 1049-331X. DOI 10.1145/237432.237434. 45

D. Zowghi and C. Coulin. Requirements Elicitation: A Survey of Techniques, Approaches, and Tools. In *Engineering and Managing Software Requirements* Aurum and Wohlin [2005], pages 19–46. DOI 10.1007/3-540-28244-0_2. 44, 46