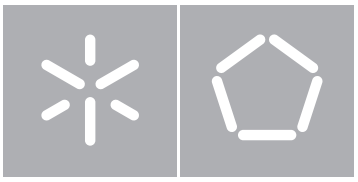
Universidade do Minho
Escola de Engenharia

Maria Madalena Pacheco Gonçalves

Guidelines for Analysis and Modelling of Reactive Software Systems

Janeiro de 2013



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Maria Madalena Pacheco Gonçalves

Guidelines for Analysis and Modelling of Reactive Software Systems

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor João M. Fernandes

Janeiro de 2013

Acknowledgements

Firstly I would like to thank Professor João Miguel Fernandes, for all the help and encouragement he gave me throughout the research and writing of this dissertation. Thank you for being a dedicated mentor and for always being available to clear my doubts.

Secondly, I thank my parents, brother and sisters, for their endless love and support in every moment of my life.

I also wish to express my thanks to Professor José Creissac Campos, and to all that, somehow, contributed to my success in this so important step of my academic career.

This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-EIA/116069/2009 and by FCT, under the grant with reference UMINHO/BI/55/2012.

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

Abstract

Reactive software systems are distinguished by their ability to interact with the environment in which they operate. Their behaviour is affected by a finite set of events that change the system global state. Such systems can be divided into three components: Controller, Users and Physical Entities; this division allows to model the particular behaviour of each component separately.

Coloured Petri Nets (CPNs) are a modeling language suitable for behavioural modelling, thus it can be used in the context of these systems. Among several advantages (and some shortcomings), they allow CPN models to be simulated and the formally verified.

This dissertation presents a set of guidelines for analysis and modeling of reactive software systems. The guidelines suggest how different components of a system can be identified and characterised. The guidelines also recommend various alternatives for modelling the system components with the CPN modelling language.

The guidelines are illustrated with a practical example, which is modelled by means of CPN Tools, a tool for designing CPN models.

The application of the guidelines allows CPN models specifically targeted for reactive software systems to benefit from executability, modularity, parameterization, and configurability.

Resumo

Os sistemas de *software* reativos são caracterizados pela sua capacidade de interagir com o meio em que se inserem. O comportamento dum sistema deste tipo é influenciado por eventos que, quando ocorrem, alteram o estado global desse sistema. Tais sistemas podem ser divididos em três componentes: Controlador, Entidades Físicas e Utilizadores; o que permite modelar separadamente o comportamento que caracteriza cada um desses componentes.

As Redes de Petri Coloridas (RdP Coloridas) são uma linguagem de modelação adequada a sistemas com uma significativa componente comportamental, pelo que podem ser usadas no contexto dos sistemas de *software* reativos. Entre diversas vantagens (e algumas limitações) elas permitem que os modelos sejam simulados e formalmente verificados.

Este trabalho apresenta um conjunto de diretrizes de análise e modelação de sistemas de *software* reativos. No processo de análise, sugere-se como podem ser identificados e caracterizados os diferentes componentes de um sistema. No processo de modelação, recomendam-se várias formas de modelar cada componente com RdP Coloridas.

As diretrizes são ilustradas com um exemplo prático, o qual é modelado com o auxílio da ferramenta de desenho de RdP Coloridas, *CPN Tools*.

A aplicação das diretrizes no contexto referido permite obter modelos que beneficiam de executabilidade, modularidade, parameterização e configurabilidade.

To my beloved family.

Contents

List of Figures

Chapter 1

Introduction

Prologue

This chapter presents the subject of this dissertation. The issues related to analysis and modelling of reactive software systems are described, and the problem underlying this research is explained. Afterwards, the motivation, overall aims and contributions are detailed. Finally, the structure of this document is presented, with brief summaries of each chapter.

1.1 General Introduction

Nowadays, reactive software is becoming very popular and gaining much importance for both users and developers. With the exponential growth of ubiquitous and interactive technologies, the critical issues, such as responsiveness and safety, are no longer the only ones that matter in the production of this software; usability is earning more and more interest in industry, with each day that goes by.

A reactive software system is characterized by maintain an ongoing interaction with its surrounding environment, responding to stimuli from that environment, and changing it with those responses. The events that stimulate responses in a reactive software system can occur at any moment, even when the system is busy responding to earlier events. In the viewpoint of the

system development, it must be decided which stimuli have to be answered, and what are their priorities; some events might be more relevant than others, and sometimes it may be necessary to interrupt an event processing that is already running, or to process multiple events in parallel. Therefore, the development of reactive software systems requires a detailed study, not only of the software itself, but also of its environment, which is composed by the people who interact with the system and by the physical objects that enable such interactions. Model-driven development (MDD) seems to be a feasible and good methodology for tackling that study.

In the context of software engineering, the concept of *model* is defined as *an abstraction of a system*, rather than just being a representation of the system [46]. Such definition entails two problems: the first resides within the decision of what must be included and what must be left out of a model; and the second concerns the decision of which approach (or approaches) must be used for modelling a particular system.

Modelling is often associated with the design and specification of models; *models*, *diagrams*, and *specifications* are sometimes used interchangeably, but there are differences between those terms. Modelling is a conceptual task, concerned with bringing different concepts and ideas together, according to a certain rationale; designing is about creating the visible or tangible artifacts (such as diagrams) for representing those ideas; creating specifications relates to describing certain desired properties [49]. In a nutshell, a diagram is used for exhibiting the specifications of a model. Despite this, *model* (*modelling*) is referred in this work, as both the conceptual and the tangible artifact (task).

Regarding the modelling of software, models serve many purposes: refining the understanding of a problem, improving the communication between peers, abstracting from complexity, finding potential solutions, among so many others. In this dissertation, models are a means to develop system specifications, from which, executable prototypes can be derived.

In [7], a prototype is defined as:

> a system *which simulates the important interfaces and performs the main functions of the intended system, while not being necessarily bound by the same hardware speed, size, or cost constraints.*

This work is concerned with the development of an approach (viewed as a set of guidelines) for modelling reactive software systems. The focus of interest is on controllers, which are systems that control the *application and*

its interactions with the outside world [40]. The Coloured Petri Nets (CPNs) modelling language is the target of that approach; however, other modelling languages, such as UML and other variations of Petri Nets (PNs), are also addressed, in order to compare different modelling techniques, and discuss advantages and disadvantages of using CPN.

The end result of applying such guidelines is a system specification, in which the models are tools for studying the accuracy and reliability of the modelled systems. Resorting to formal analysis and simulation, one can reach a range of dimensions that prototypes based on very high-level languages or fourth-generation languages [46], can not address so easily. These prototypes will be useful to study two dimensions of a reactive system: the system itself, as a cluster of computer functionalities, and the environment, where that system is integrated. Within that environment there are the users (also called human actors), whose behaviour must be included in the model.

This work was developed in the context of an academic project called *APEX - Agile Prototyping for user EXperience*, which aims the study of user experience, by creating executable prototypes, and providing the users a virtual interaction with systems that are not yet (fully) developed. Although the present work does not focus on the study of user experience itself - the actual deployment of executable prototypes and enforcement of virtual simulations are not addressed - it is a means towards that end, by presenting a set of guidelines for the design of models that can easily be adjusted to serve as executable prototypes.

Allowing the transformation of information models into executable prototypes requires four main features to be achieved by those information models: model execution, modularity, parameterization and configurability. Model execution is a step closer into making the prototypes executable, whereas the remaining features enable the prototype management.

This work has the following intended audiences:

- the general software engineering community, and more particularly software architects, analysts, requirements engineers, and modellers;
- those concerned with the modelling of reactive software systems and controllers, or even with other systems that share common features with the latters, like ubiquitous, interactive, embedded, or real-time systems;
- anyone who is interested in modelling with the CPN modelling lan-

guage;

- those interested in guidelines for both analysis and modelling of software.

1.2 Problem Statement

The need for some methodology in the modelling of software has been supported by numerous authors and is an issue of research [42]; the problem resides in finding which features are common to the majority of cases. However, it is not easy to detect such properties within general software; for example, the phrase *software systems* comprises, by itself, a set of information so very wide and branched, that it seems almost impossible to find out any set of rules that can be applied to its entirely. In addition to this situation, every modeller has its own opinions and practices, which makes every model a representation of its modeller's favourite viewpoints and perceptions of the reality [42]. Although having a personal *touch* does not interfere with the correctness and adequacy of the models, nor it makes them wrongly built, when dealing with a whole team of modellers, there are issues that must be taken into account, like model comparison, analysis, composition, reuse, among so many others. Consistency between models is crucial to anyone who needs to keep his/her specifications suitable through time and *through people*. This is the reason why most software modelling methodologies apply only to specific classes of systems, instead of broad and generic groups of systems. The present work is targeted to one of those sets: reactive software systems.

To model the behaviour of a reactive system, one must consider the behaviour of all entities that may trigger a reaction from that system; an example of such entities are the system users.

Because systems are (usually) used by people, it is important to consider human behaviour when developing those systems. Understanding the users' needs is essential to adapt our systems to their demands. However, modelling the behaviour of users is a difficult task, regarding general software modelling; one can never predict the exact behaviour of people facing computer systems. In fact, that is more a subject of study for human psychology and sociology than for software engineering; nevertheless, we, as software engineers, need that information to describe human-computer interaction features and to im-

prove the user experience concerning the software we produce. The available resources to detect requirements related to human-computer interaction and user experience are still scarce, inefficient, unreliable, and expensive. Executable prototypes are a solution that can help overcoming these obstacles, by allowing users to interact with a version of a system, early in the development phase. Not having to actually deploy the system, and yet being able to test it and to study the behaviour of its users, is an appealing thought.

The guidelines proposed in this dissertation arrive as an endeavor for creating specification templates than can be easily configured to suit the needs of a system and their users. The proposed guidelines aim the design of models that are accurate enough for the development of reliable prototypes, and structured enough to keep up with future changes in the system requirements. It is the ultimate purpose of this work to create evolutive model specifications, which are capable of following the natural development of a system to be implemented.

Although it has not been proved true within this work, it is expected that allowing the users to interact with a system in early stages of its development, enables the discovery of more precise requirements, which helps developing software towards meeting the needs of its users. The idea inherent to this approach is addressed in [50] as *Early-Phase Requirement Engineering*. Another prospect is to reduce the costs (both financial and temporal) of system implementation and testing [44]. However, studying the feasibility of these expectations goes beyond the scope of this dissertation.

1.3 Motivation

The guidelines proposed in this work apply to reactive software systems that can be described as controllers, and address the modelling with the CPN modelling language.

The following three questions must be answered to help the reader understand the relevance of this study:

- *Why reactive software systems?* explains the importance of studying these kind of systems;

- *Why CPNs?* emphasizes the features of this modelling language, and shows to what extent this study benefits from them; and

- *Why guidelines?* addresses the need for analysis and modelling heuristics in software engineering.

Why reactive software systems?

Reactive software systems exhibit a set of characteristics that make them particularly challenging, when it comes to modelling and deploying them; nondeterminism, perpetuity, asynchrony, time, and concurrency are the most relevant ones [20, 21]. In other kinds of systems, such characteristics may appear in isolation, or in smaller, and therefore less complex, sets, which makes reactive software systems a kind of systems of major importance. Examples of systems that share some of these features are real-time systems, embedded systems, process control systems, device control systems, network protocols, ubiquitous systems, and user interface (UI) applications.

To model a reactive system, one must take under consideration the external events from the environment, and thereby, add them in the model of that system; the environment itself must be contemplated in the model. Sometimes, that environment includes users actions; hence, the modelling of the environment must address human behaviour.

One of the great challenges of modelling the behaviour of a person lies within the natural human unpredictability. One can never be 100% sure of what a person will do when facing a software product; much less a group (possibly gigantic) of users. The unpredictability is nearly unimaginable.

Today there are several tools and techniques for forecasting behaviour and events that might happen when using a system; predictive analytics, predictive models, and data mining are examples of that [35]. Yet, those techniques are still inefficient for unveiling requirements related to user-experience; until now, typical solutions rely with questioning the users about their likes and dislikes, and trying to find out a balance between the users' opinions.

The importance of studying reactive software systems is supplemented with the need to find artifacts for understanding, foreseeing, and describing human behaviour, within software. Nowadays, this is a need of utmost importance, because of the exponential growth of the market of systems based on human-computer interaction. Today, UI applications are a wide-world social trend; such situation calls for more efficient and reliable ways of developing such systems, than those we now have at our disposal.

Why CPNs?

The CPN modelling language is a very useful formalism for describing the behaviour of systems; it provides graphical and textual constructs to deal with synchronization, concurrency and communication [24], which are recurrent issues in today's problems. There are two main features that make this an interesting modelling language: (1) being executable, which enables model simulation and animation; and, (2) having formal semantics, which, along with numerous analysis techniques, enables validation and formal verification.

Other features are also supported by the CPN modelling language, such as hierarchy and modularity, parameterization, locality, mobility and context-awareness, among so many others [12, 13, 26, 47]. Dealing with so many important issues makes this modelling language useful across numerous application domains [18], for example: telecommunications and network protocols; distributed software systems; embedded systems; workflow management systems; hardware and software architectures; and UI applications.

Nevertheless, there are some drawbacks when using the CPN modelling language: (1) the gap between modelling and implementation - an automatic transformation of models into actual software is not an easy task within the CPNs(although it is not impossible [26,48]); (2) the lack of different diagram forms - CPNs model only data and state machines - which turns out to be very restrictive when compared to other languages, such as UML, with appropriate diagrams for different problem aspects (functional, behavioural and structural) and different groups of people (users, clients, developers, analysts, or others).

The CPN modelling language can deal with complex systems, and therefore it is widely used, both in the academic and industrial fields. It is also very good as a complementary tool for other modelling languages, including UML; a combination of both languages increases the range of model applications, and creates a new set of properties that each individual language, can not provide on its own. Some of those advantages are explored in [27], where use cases are made executable, for description and validation of requirements and specifications; and in [14], where a tool is proposed for translating some UML diagrams into a CPN, which increases the number of analysis and simulation techniques available for UML.

Notwithstanding the disadvantages mentioned above, the CPN modelling language seems to be a good option for the development of prototypes. The

choice of using CPNs in this work is due to the existence of good simulation and analysis tools. The language is rigorous enough to compare and assess different models, which allows the study of different scenarios, and helps choosing the one that best suits the requirements of a particular problem. In the development of executable prototypes, those are interesting and valuable features.

CPNs are supported by *CPN Tools* [1], a graphical editor, that allows model visualization and animation, also providing tools for analysis and simulation. *CPN Tools* is vendor independent and cost free, and it is licensed to thousands of users, both in the industrial and academic fields.

Why guidelines?

First of all, it is important to state the meaning of "guideline":

> *A principle put forward to set standards or determine a course of action. [1]*

It must be emphasized that guidelines are no more than suggestions, therefore, they are never mandatory nor enforced.

In software engineering, more particularly, in software development, the existence of guidelines for analysis and modelling of a particular piece of software is not senseless. Guidelines ease these tasks because they highlight the patterns within them.

According to [38], patterns are *already proven conceptual solutions to recurring problems*. This means that they avoid the search for solutions of problems that appear frequently. Patterns not only highlight a recurrent problem, but they also present an already proven solution for it.

Nowadays, many patterns were identified, for both analysis [15] and design [17] of software. It is unreasonable (and a waste of time and resources) not to use them when they are applicable, but unfortunately, there are not enough patterns in the world to solve all the software problems that a software analyst, modeller, or developer comes across (or at least, those patterns were not identified yet).

Guidelines, may not be problem solvers, like patterns are, but they are useful tools to find viable solutions. The quality of the solutions will depend on whether the guidelines are being properly or erroneously applied. Hence,

[1] CPN Tools website: `http://cpntools.org`

it is important to establish precisely the scope of the guidelines, i.e., in what circumstances they can be applied. The broader the scope of the guidelines, the harder it is to ensure the usefulness and reliability of the guidelines. One must find a balance in the guidelines: these can not be so thorough that they can only be applied to a so very specific (and perhaps useless) context, but they must be detailed enough to be applied to a set of examples that share particular features.

Two main advantages arise from the usage of guidelines in software modelling:

- Models formal proof of adequacy: if the guidelines ensure a set of desired formal properties, and if the models are based on such guidelines, then it is easy to prove the adequacy of those models [33];

- Models standardization: the models of different problems or different views of the same problem become more similar to each other, which eases their comparison (it is easier to compare models built under the same approach and guidelines, rather than comparing models based on different groundwork) and avoids ambiguities. Different modellers usually use different approaches and languages, because of their professional experience and their personal *tastes*; even if their final results are similar, it is hard to compare the models and choose the one that best suits a particular situation, if those models were built under different approaches. In the words of M. Fowler [15]: *Models are not right or wrong, they are more or less useful.* Model standardization eases the discovery of the most useful models.

1.4 Contributions

This section presents the contributions of this dissertation to the software engineering community.

The main contribution of this dissertation is the systematization of the modelling process for reactive software systems. Guidelines for analysis and modelling of reactive software systems are proposed as part of a process for the development of executable prototypes. The ultimate purpose of such prototypes is the development of a tool for studying user experience.

Much of the content of this dissertation was developed as part of an

academic project, called *APEX - Agile Prototyping for user EXperience*[2], which demonstrative example, a Smart Library, was adopted as an example of this dissertation, to exhibit and explain the outcomes of this study.

During the writing of this dissertation, a paper was developed and presented in a conference workshop, to discuss the results available at the moment, and to acquire feedback on the proposed approach. The contents of that paper are reflected in this dissertation. A reference to that paper follows next:

> Madalena P. Gonçalves and João M. Fernandes. *Guidelines for Modelling Reactive Systems with Coloured Petri Nets.* In 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2012) at ASE 2012, volume 7706 of Lecture Notes in Computer Science, pages 126–137. Springer, Essen, Germany, Sept 2012. In Press.

1.5 Overview

This dissertation is organized as follows:

Chapter 2 discusses behavioural modelling. It is given a brief comparison of three modelling languages that excel in the behavioural aspect: Statecharts, behavioural and interaction UML models, and Petri nets. Emphasis is given to the CPN modelling language, so its structure and properties are explained in more depth, than the remaining modelling languages.

Chapter 3 presents the Smart Library, the demonstrative example that is used in this work to illustrate the practical application of the proposed guidelines.

Chapter 4 presents the proposed guidelines. These are divided in two processes: initially, the analysis of the problem is addressed, describing the relevant components of a reactive software system, and explaining how those components can be recognized and dealt with; afterwards, it is discussed the modelling of reactive software systems with the CPN

[2]The APEX web site: `http://wiki.di.uminho.pt/twiki/bin/view/Research/APEX/WebHome`

modelling language, and with *CPN Tools*. The solutions outlined in the guidelines are explained by means of practical examples.

Chapter 5 draws the final conclusions of this research, and gives some notes on how this work can be further developed.

Chapter 2

Modelling Behaviour

Prologue

This chapter compares some behavioural models, namely Statecharts, UML behavioural models, and Petri nets (with their numerous variations). An historical perspective is provided about the evolution of such models. Afterwards the CPN (a variation of Petri Nets) is reviewed, explaining its structure and properties, and also addressing simulation and formal analysis of the CPN models.

2.1 Introduction

Software analysis and modelling is about discovering the functional, behavioural, and structural aspects of a system [21], and creating reliable representations of those aspects. Functional aspects define what the system is supposed to do, which actions or events must be performed to fulfill a set of requirements, and how the information flows in the system. Behaviour is about dynamics, defining order and time constraints for the occurrence of the system actions; finally, structure defines the system architecture, which are the system components, how are these organized, and how they communicate with each other. Throughout the years, several studies have been made in order to discover better ways to tackle all these aspects, and great

efforts have been conducted to achieve better, simpler, and faster tools and languages for software modelling.

In the modelling of reactive software systems, the focus turns mainly to the description of behaviour, and the attempt to achieve a representation of it (i.e., an abstraction) that simultaneously meets all the system requirements, and resembles, as closely as possible, the reality. Thus, it is necessary to understand what behaviour is, what are the inherent challenges to its representation, and what mechanisms exist to overcome or work around such issues.

Definition 1: Behaviour (in software)
Behaviour is the set of actions which a system, or a system component, performs in a specific order, and sometimes obeying time (and/or other) restrictions [3]. Behaviour can either be discrete or continuous; in the latter, system operation takes in consideration previously performed actions, while in the former, there is no record of past occurrences [9].

One of the greatest difficulties in modelling real-world behaviour relates to it being continuous. However, modellers can overlook that continuity by assuming as important just some parts of that behaviour [49]. This discrete-event perspective assumes that a finite set of events is enough to describe the general and relevant behaviour of a particular system.

Definition 2: Event
An event is a significant occurrence at a moment in time [31, 40], which may trigger the change of states. In the case of reactive software systems, events may incite several entities of a (composite) system to produce a predefined response, and they also may encourage users to perform some desired actions. Events may be external or internal to an entity. An *external event* is an event that occurs among entities; it is caused by an entity, and may affect several others. The occurrence of such events can be recognized by all entities within the composite system, and may affect them. An *internal event* is an event that occurs within an entity; only that entity is affected by it, and the remaining entities are not aware of such occurrence.

In addition to the discretization of continuous behaviour, a lot of other assumptions help handling the inherent complexity of reactive software systems. Describing a system as a finite set of states is one of them.

Definition 3: State
A state is a system (or entity's) condition at a moment in time, where that moment corresponds to the time between events [31].

In this assumption, a system always finds itself in a certain state, which mirrors a clear and stable system condition. When an event happens (which is often considered to be an atomic occurrence) it takes that stable condition into an unclear and unstable one, called a transition. Transitions are only temporary conditions, and when a transition terminates, the system re-assumes a stable condition.

Definition 4: Transition
A transition is a system (or entity's) condition that can be observed in the time between a change of states [49]. It is caused by the occurrence of a particular event and it usually sets off an action or sequence of actions.

This work is concerned with the modelling of discrete behaviour; from what has been discussed so far, it is clear that one can describe the behaviour of a reactive system as a set of events, states, and transitions. But this is just a way of doing it. There are numerous languages and models that allow to represent software behaviour, and these can be split in three categories [5]:

Control flow models These models emphasize the order in which tasks occur; relevance is given to knowing who *has got* the control of the system, in a particular moment, who had the control just before that, and who will have right after. These models are not suitable to represent flow of data.

Data flow models These models emphasize the flowing and manipulation of data; relevance is give to knowing the source and destination of data, and to the changes they suffer during that flow.

State Machines These models emphasize the responsiveness of a system to external stimuli, i.e., they focus on the system reactions to particular occurrences in the environment.

In the following sections it is provided a brief overview on available behavioural models, addressing Harel's Statecharts [20, 21], UML behavioural models, and Petri Nets (PN).

2.2 Some Behavioural Models

2.2.1 Statecharts / State Machines

The modelling of reactive software systems found its first great development with Statecharts, which are structured state machines. State machines are used to describe a system through a finite set of states the system can assume. In Statecharts, those states are split in groups representing different pieces of the system, and the joining of those groups makes the whole modelled system. Statecharts are suitable for describing event-driven discrete behaviour, and also for the modelling of concurrent systems [19].

The problem with statecharts (and any state machine) is that the number of states increases quite quickly, and representing large, complex systems becomes difficult and error prone [46]. This is known as the *state explosion problem.*

2.2.2 UML

The UML was born in the early 90's, from the contribution of Booch, Rumbaugh, and Jacobson (known as the *Three Amigos* [16]) due to the need of creating a modelling standard that was the unification of the numerous object-oriented modelling techniques that existed at the time. The high number of different modelling approaches was hampering the choice of which method to use in each situation; companies often needed to create their own analysis and design techniques, which made the integration of different techniques too complicated. In 1997 it was adopted by the Object Management Group (OMG), and since then UML has been evolving to better tackle the problems inherent to systems modelling.

The UML is nowadays widely used in industry, and provides a set of different diagrams for the modelling of behaviour, which are shortly described next:

- **Use Cases**, for describing the system functions and the interactions between actors and the system;
- **Activity Diagrams**, for describing control and data flow;
- **State Machines**, for representing reactivity;

- **Sequence Diagrams**, for describing how interactions happen, based on their time and order of occurrence;
- **Collaboration Diagrams**, also for describing how interactions happen, based on the structure of the interactions.

2.2.3 Petri Nets

In 1962, Carl Adam Petri developed the first version of what would become known as Petri Nets [6], a powerful mechanism for modelling the behaviour of asynchronous distributed systems. Combining a graphical notation with well defined formal rules, he was able to model computer systems in the same way other scientific areas represent their models: relying on physical and mathematical bases.

Many ideas and modelling techniques were developed around the first *draft* of PN, such as process algebras (another formal technique to model concurrency), and several classes of PN. The most important are described below, in the order they emerged. It is noteworthy that each of the following nets is an evolution of the previous ones, that arose to introduce new concepts and ideas to the theory that had been developed up to that moment.

Place/Transition Nets (PT-nets) The very first approach following Petri's Condition/Event Nets (CE-nets), where models are represented as a graph, with places and transitions as nodes. In the places are stored tokens which are caused to flow to other places by the occurrence of transitions. At each moment, the positions of the tokens in the graph describe the global state of the model. Tokens are indistinguishable, and the nets represent low-level models.

Coloured Petri Nets (CPN) While PT-nets used *black* tokens, the CPN brought colours to them, introducing the concept of token types (the so-called *colour sets*). With such an abstraction it was possible to distinguish the tokens and manage them more easily. The nets could then represent high-level models. Also, a new functional language called *CPN ML* (which was based on the Standard ML) was developed as a complement to the graphical notation, to allow to formally define and manipulate the colours.

Hierarchical CPN Also known just as CPN, upgraded the old CPN with more powerful structuring mechanisms. Not only the models could be

separated into different modules, it also became possible to organize the models in an hierarchical structure, where those modules had well-defined interfaces that enabled communication and exchange of data between them. Nets became suitable to represent high-level models.

Object Oriented Petri Nets It is an idea that is still under development; although many approaches have already been developed [4, 29, 30, 32, 34], none has yet been able to reach a level of stability and reliability, that would allow them to be accepted as a new modeling language. Despite the variety of techniques, there is an idea that is shared by them all: relying on the foundations of the object-oriented paradigm. Inheritance, polymorphism, and encapsulation are just a few examples of OO concepts that are lacking in older versions of PN and that are being pursued by the several investigators in order to make the PN mechanism better and more suitable for industry-level systems.

After this short overview on the history of modelling techniques, the CPN is more thoroughly addressed next.

2.3 Coloured Petri Nets

Coloured Petri Nets (CP-nets or CPNs) is a modelling language that allows its models to be executed, by combining a graphical notation with code primitives. With executability comes model simulation, which is one of the main features of this language; it helps testing the models, making it easier to identify errors, as well as finding forgotten or unpredicted aspects of the modelled systems. Another important feature of this language is that it was built on mathematical foundations and has an explicit formal description; with strict syntax and semantic rules, CPN models can be submitted to validation and formal verification, which allows one to check the fulfillment (or not) of user requirements, and to reason about the correctness of the models, respectively.

The coding primitives are declared in *CPN ML* (CPN Modelling Language).

These nets can be applied in several application domains, making them suitable for modelling industry level systems [25]; they provide primitives for dealing with concurrency, synchronization, and communication issues,

which are often present in complex computer systems. The CPN modelling language is also able to deal with hierarchy and time.

CPN is also very good as a complementary tool for other modelling languages, such as UML; using these languages together it is possible to exploit the properties of each language and gain leverage from the set of new properties that arise from such combination. Several authors have addressed the particular combination of UML and PN (or some variation of PN), and studied its benefits and drawbacks in [10, 12–14, 27, 37, 41].

2.3.1 Structure and Concepts

A CPN is represented by an oriented graph with two types of nodes: **places** (drawn as ellipses) represent states, and **transitions** (drawn as rectangles) represent events. In these graphs, adjacent nodes can never be of the same type; it is not possible for two places or two transitions to be directly linked, which implies that a change of states can only happen through the occurrence of an event, and an event can only occur to toggle a pair of states.

Places can hold *bags* of tokens of a specific type. A CPN model, a token represents an instance of something (an object or entity, for example), and its type is called a **colour set**. The *bags of tokens* can contain several appearances of the same token, thereby it is said that places can hold **multisets** of tokens, rather than just *sets* [23].

Colour sets (the tokens types) can either be simple or compound, similarly to what happens with types in general programing languages. The simple colour sets represent the most basic types (like Integer, String, and Boolean), and the compound color sets are built as a combination of some simple colour sets (like Lists, Products, Unions, and Records).

Each token stores a particular data value that matches its colour set characteristics. Tokens travel through the models by the occurrence of transitions, carrying data from one place to another, until a desired model status or requirement is reached. At any given moment, a CPN model has a global state, which is equal to the set of all tokens and their locations in specific places, at that moment. Such global state is known as the model **marking**, and the very first global state of a model is called the **initial marking**.

In addition to the nodes and arcs, CPN models contain code primitives and inscriptions for handling data within the models. The **code primitives** can be: definitions (such as the colour sets), variables (tokens with predefined values), conditions guards (that evaluate whether a transition can or

cannot occur in any given time), or functions (to manipulate data values). Each graphical form has its own **code inscriptions**: places have a name, a colour set, and an initial marking; transitions have a name, a guard, a time delay, and a code segment; arcs have an expression that represents the data value that is being transported in that arc. The arc expression can be a function or a variable, evaluating either to a multiset or to a single value; anyway, such expression always matches the colour set of the place to which the arc in connected.

The execution of a CPN model translates into a series of transition occurrences, allowing tokens to flow within the arcs, from place to place. The match of a token to an arc expression results in a **binding element**. If there are at least one binding element in each input arc of a transition, and if the condition guard of that transition evaluates to true for those particular binding elements, then the transition can occur, executing its own code segment, and placing tokens (resulting from the execution) in its outgoing arcs.

In the same step of execution there can be several transitions ready to be fired (i.e. ready to occur), and they can either be in conflict or in concurrency with each other. Two or more transitions are in **conflict** if the resources (tokens) existing in shared input places, are not sufficient to satisfy the occurrence of all the transitions in that step. The occurrence of one of those transitions disables the remaining, because there no longer exist binding elements for those transitions. Enabled transitions that are not in conflict (either because they do not share input places, or because there are enough tokens for all binding elements) are called **concurrent**; one transition can occur without disabling the others. When there are several enabled transitions, the choice of which one is fired next is random; this is what makes CPN models non-deterministic.

CPN models can be constructed in various modules, each representing specific parts of a system, that can communicate with each other and exchange data through well defined interfaces. Hierarchy can be depicted in CPN by three graphical mechanisms: substitution transitions, that implement modularity, and port places and fusion places, that implement interfaces. **Substitution transitions** are drawn as double-edged transitions and they are a reference to an external module (a **submodule** or **subpage**). Such transitions do not work like the normal ones, because they can not become enabled, therefore they never occur; they represent a general system operation which is detailed in a different module. Port places and fusion places are two different ways of creating communication interfaces: they rep-

resent places that are shared by different modules and through which tokens can be exchanged. A **fusion set** represents a conceptual place that is the merge of several **fusion places**; these places always have the exact same state, which reflects itself in the state of the fusion set. Fusion sets can be shared among many different modules, working in some sense like global variables in general programming languages; therefore, caution is advised in the handle of data within these sets. A **socket** represents a communication channel (either unidirectional or bidirectional) shared by *only* two socket places. **Socket places** are the input and output places of substitution transitions, and they are merged with port places through **port assignment**; the **port places** are the representation of socket places in each individual submodule. A port place can either be an *input place*, and *output place*, or an *input/output place*. Sockets are private channels and their contents can only be known by the respective port places; additionally port places cannot be shared (like fusion places are), therefore sockets represent a more secure interface mechanism than fusion sets.

The hierarchy in a CPN model can reach several levels of depth, and there can be instances of the same modules running at the same time in a CPN model, each with its own marking, and independent from other module instances.

Apart from the just described purpose, fusion places can also be as a synchronization mechanism; the CPN also supports transition synchronization, but this feature is not supported by the designing tools currently available.

2.3.2 Tool Support

The modelling with CPN is supported by *CPN Tools*, a tool for designing and editing CPN models. It provides a graphical user interface for building the models, a simulator for model testing, as well as several analysis tools, for performance and state space analysis.

Besides the simulation and analysis mechanisms, this tool also performs quick model checking, at construction time, and provides visual error messages about syntax and type errors, which is very useful, mainly because they usually are innocent mistakes, that can be so small, that they go undetected, even to the eyes of more experienced modellers.

This tool has an open architecture and there are several libraries that allow to extend functionality [24], enabling modellers and investigators to create auxiliary tools for specific areas and purposes, at their own will.

CPN Tools is vendor independent and cost free, and it is licensed to thousands of users, both in the industrial and academic fields.

Simulation

Model simulation can be used for model testing to find forgotten, unpredicted, or unnoticed errors, but it is also very useful for gaining an overall knowledge of what a system is, what it is supposed to do, and how it should work. Thus, the simulation tool is suitable for explaining a modelled system to people that are not familiarized with it.

Simulation can be either interactive or automatic in *CPN Tools*: in **interactive simulation** the user can choose, in each step, the binding elements and the transition to be fired; while in **automatic simulation** this is done automatically. In the latter, both the binding elements and the transitions to fire are chosen randomly by the tool.

For each performed simulation, *CPN Tools* creates a **simulation report**, which is a text file containing all the information about a simulation: steps, time, fired transitions, chosen binding elements, and other relevant data.

Performance Analysis

Another important feature of CPN is the handling of time: CPN can be extended with a model time concept (a feature that CPN inherited from previous PN versions).

It is important to acknowledge the difference between real time and model time, because they are not the same; **model time** is related to simulation and it is virtual time, measured as model steps, by a global clock that is associated to the model. It does not have any sort of relationship with real-time.

In a timed CPN, tokens can carry a **time stamp** indicating the time they are ready to be used by an occurring transition; In a timed net, a multiset becomes a *timed multiset*, a marking becomes a *timed marking*, and so on, making all untimed CPN concepts adapted to handle time [24].

Timed nets are a mechanism, supported by *CPN Tools*, for studying the performance of a model in terms of operation efficiency, correct timing of events in real-time systems, and fulfillment of requirements deadlines, also in real-time systems [25].

CPN Tools provides one other tool for analyzing the performance of a model, called **monitors**; these monitors can *observe* a simulation and perform pre-defined actions if certain pre-defined conditions are verified during that simulation. Inspecting places, counting how many times a particular token value is used, and writing in external text files a customized simulation report are just a few of the things monitors let a modeller do.

Formal Analysis

Every concept explained so far, has a formal definition in CPN; all together become the formal foundations of CPN, which makes this modelling language so reliable. Due to this, it is possible to perform formal verification of a CPN model, and analyse its correctness.

The *CPN Tools* provides a tool for constructing state spaces and analysing some model properties. A **state space** is a reachability graph which nodes represent all the reachable markings, and the arcs represent the respective binding elements. The tool can generate a **state space report**, a text file with statistical information from the state space and some other detailed information about behavioural properties of the model.

The most common properties in CPN models are:

- **Reachability**, that calculates all the possible reachable markings;
- **Boundness**, that calculates the maximum and minimum number of tokens and of multi-sets that a place can hold;
- **Fairness**, that calculates how often a marking occurs in infinite occurrence sequences;
- **Home**, that calculates the markings that can be reached from a given marking;
- **Liveness**, that calculates the existence, or not, of dead markings, where no transitions are enabled (i.e., where the simulation terminates).

These properties are fully calculated by the tool, otherwise the formal analysis of a CPN model would be impracticable for large, complex systems. Each property being studied can be translated into *CPN ML* **query** functions, that allow such computation. In addition to the just described properties, other properties can be analysed, but the modellers have to write the respective queries.

Summary

Statecharts, UML, and Petri nets are three modelling languages that can be used to model behaviour. Statecharts evolved within the modelling of reactive systems however, they are not good enough to tackle the complexity inherent to today's problems. UML provides several models for addressing different system aspects, from describing requirements, to describing interactions. Yet, the features of Petri nets (known today as CPN) are, in the context of this work, more appealing than those from the other languages. The CPN modelling language has techniques for formal analysis, performance analysis, and model simulation, which are good tools to *have at hand* when developing prototypes.

Chapter 3

The Smart Library Practical Example

Prologue

The Smart Library example is presented in this chapter. It serves to illustrate the applicability of the guidelines proposed in this work.

3.1 Informal Problem Description

The Smart Library is a system that recognizes library users and guides them to their requested books. The request of a book is not an action of interest in this problem. Only registered users are taken into account. Presence sensors, scattered throughout the library, are responsible for real-time recognition of users and books locations (inside the library or in its surroundings). Users carry an id card or device (like a PDA), that makes them recognizable by the sensors. Books have RFID tags for that same purpose. Users also carry an electronic map of the library, that shows them, in real-time, the paths to their requested books (if the books are in the bookshelves). As the user follows the path suggested on the map, the path is updated according to the user's new position. When the user approaches the book he/she is looking for, lights of a specific and unique colour are turned on, highlighting the book and the bookshelf where it stands. Different light colours are used to

distinguish the requests of users in nearby areas. If the user is entering or leaving the library (through entry and exit gates), a screen near the gate displays a list of the users requested and returned books [45].

3.2 Informal Description of Specification Decisions

From the description above, the Smart Library can be seen as a set of the following Physical Entities: Books, Presence Sensors, Lights, Gates, Displays, and PDAs (or some similar device). Another entity, that is not mentioned in this description, is needed: Pressure Sensors.

The purpose of each of these entities is explained below.

3.2.1 Books

Books can be picked up from bookshelves by Users. Each book has an RFID tag and a light. The light is turned on when a User, looking for that book, is near it; and the same light is turned off when the User goes away from the book. If the book is not on its shelf, the light is not turned on (even if the User is looking for it).

3.2.2 Presence Sensors

Temperature sensors are used to detect the presence of Users inside different areas of the library. Each sensor has a sensing area and a set of devices that are inside that area, and therefore, get affected by the readings of that sensor. Each temperature sensor is triggered by: (1) the movement of Users inside its sensing area; or by (2) the absence of movement in its sensing area for a short period of time.

3.2.3 Lights

There are lights in the bookshelves and in the books. Lights can have different colours and can either be turned on or turned off.

3.2.4 Gates

There are gates for entering and leaving the library. Gates can either be open or closed.

3.2.5 Displays

Displays can either be showing a default message or a non-empty list of User's information (such as name, and requested and returned books). Displays are connected to gates, so when a User enters or leaves the library, his information can be added to the display.

3.2.6 Pressure Sensors

Pressure sensors are used to detect the presence of books on bookshelves. These sensors are located on the bookshelves and can identify a book by its RFID tag. A pressure sensor becomes active if there is a book on top of it, and becomes idle, if there are no book is on top of it.

3.2.7 Local Positioning Devices

Each User carries a device that not only serves as identification within the library, but also shows a map of the library, identifying in real time, the location of the User, and the locations of the books the Userhas requested.

There are two more entities to consider beyond the Physical Entities:

3.2.8 Users

Only registered Users are considered. Users can move inside the library (or in its surroundings) following a map that shows, in real-time, their location and the location where their requested books are stored. Once a User finds a requested book, he/she can pick it up from the bookshelf. The returning of books is also considered, however, the request of books is not.

3.2.9 Controller

The Controller makes the bridge between all the communicating Physical Entities. These send their outputs to the Controller, which processes the

received data. If the data is meant to another Physical Entity, the Controller redirects it (either as it was received, or translated into a notation that the destination entity understands).

Chapter 4

Guidelines

Prologue

This chapter presents the guidelines proposed in this dissertation for analysis and modelling of reactive software systems with CPN modelling language. Some notes about the guidelines are first discussed, explaining their goals and purpose, and how they can be used. Afterwards the guidelines are presented, illustrated with practical examples from the Smart Library.

4.1 Modelling Approach

According to [13], controllers can be described by an architecture of three components - Controller, Physical Entities, and Users - as the one shown in fig. 4.1. The Controller is the part of the system to be developed, the Physical Entities are real life objects, which are embedded in the system, and the Users are the people that use the system. These three individual components are connected to each other through well-defined interfaces (referred in fig. 4.1 as A and B).

This structure implies that controllers are described by means of a *system* and its *environment*; while the system is composed of the Controller and the Physical Entities, the environment comprises the Physical Entities and the Users. The users face the entire system only as the Controller and the

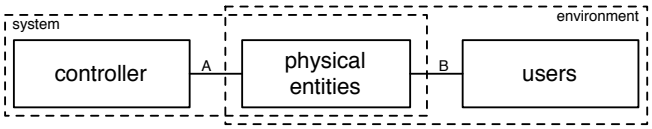


Figure 4.1: General architecture of reactive software systems (This image was copied from [13])

Physical Entities, without always acknowledging the separation of these two components; they usually know only the interface B, which allows them to interact with the system. A developer *sees* more than this, acknowledging the environment also as part of the global system.

This component-based architecture allows to describe the behaviour of each component separately. On one hand, the Physical Entities represent real world objects, so their behaviour is already defined, which usually prevents the modellers of changing it; these entities must be handled as they are, and not as the modellers wish them to be. On the other hand, modellers do not know, at the outset, the behaviour of the Controller and the behaviour of the Users; thus, they can model such behaviours as they expected them to be.

Fernandes et al. also suggest, in [13], the use of scenarios to model the behaviours of the Controller and of the Users, due to the tendency of these entities to undergo several changes throughout all the system development process (which includes analysis, modelling, and implementation).

The guidelines proposed in this chapter address the modelling of two aspects of a system: architecture and functionality. In architectural terms, an Object-Oriented (OO) approach is taken: the system is split in individual components, each with its own properties and logic. Such components can communicate with each other, by means of well defined interfaces. In functional terms, there is an attempt to streamline the behavioural description of each component, proposing solutions for different possible situations. Throughout the very description of the guidelines, the different ways of analysing and modelling the functional aspects of each component are studied. The several perspectives presented in the guidelines are explained with practical examples, exposing the differences, advantages and disadvantages among the different options.

The Smart Library example (recall chapter 3) is used to demonstrate the applicability of the guidelines. It is important to notice that the main

purpose of this example is to apply rapid-prototyping in the study of user behaviour towards the system; the decisions made for the modelling of this system make sense in the context of this problem, but they may not be the most suitable in other cases. These guidelines focus on goals that may not match the goals of other problems, thus they should be seen, not as a final and unique solution, but as a possible one, which can be adapted according to the context and purpose of each problem. The guidelines address both the analysis and the modelling of reactive software systems, with the CPN modelling language, and there is no specific order to apply them, except that (some parts of) analysis must come before (some parts of) modelling; besides that, there are no objections to customizing the guidelines, or even skipping some of them. The order in which they are presented made sense in the demonstrative example where they were applied, but nothing indicates their order cannot be changed, according to the modellers' preferences.

Four main goals were the basis for the development of the guidelines:

- the creation of interactive prototypes;
- simulation and formal analysis of models;
- model testing; and,
- management of problem complexity.

The assumptions made during the modelling of the Smart Library are based on the just described goals. The most evident assumption within these guidelines, refers to the communication between physical entities; all messages exchanged between these entities must pass by the Controller before being forwarded to another entity. This may not be the most advisable solution in real situations; the actual objects (the ones that are represented by the Physical Entities in the models) may be physically connected to each other. However, it is the intention of these guidelines to create models that can quickly be modified, thus, it is easier to ignore the real connections and assume that all the objects are connected to the Controller, which filters received messages, interpreting and rerouting them to their target objects. By making this assumption, the Physical Entities become structurally independent from each other, which means that changing the model of a Physical Entity does not require changing the models of other Physical Entities, even if any sort of connection exists between them. This allows performing rapid

changes in the models, which is a desired quality in prototyping. Other abstractions are presented and explained throughout the very description of the guidelines.

Before moving on to the next subject, it must be pointed out that guidelines can make the processes of analysis and modelling less laborious, but they cannot do all the work. Finding out which properties and functionalities are important in the development of a system may require a lot of effort; therefore it is advisable to apply the guidelines carefully, without forgetting the context and goals of the problem.

The following guidelines refer mainly the project modellers, but not with the intention of disregarding the remaining people involved; on the contrary, it is assumed all the stakeholders can participate in the decisions related to the problem and to the models, specially during the process of analysis.

The guidelines are listed below, in order to give to the reader an overview of those guidelines, so he/she do not get lost in the thorough description that follows in the next sections.

Analysis Guidelines

1. Identify the Physical Entities
2. Identify the Users
3. Identify functionality and structure
   i. Identify the functions of the Controller
   ii. Identify the actions of the Users
   iii. Identify the communications net
4. Identify the private phenomena of the Physical Entities
5. Identify phenomena shared between Physical Entities

Modelling Guidelines

1. Create pages
2. Draw the Physical Entities
   i. Draw States
   ii. Draw internal events
   iii. Set the data flowing direction
   iv. Declare colour sets

   v. Declare *CPN ML* primitives

3. Draw interfaces for shared phenomena
4. Draw scenarios
   i. Create initial values
   ii. Create the desired behaviour of the Controller
   iii. Create the desired behaviours of the Users

4.2 Analysis Guidelines

Any modelling technique must always be preceded by a thorough study of the problem, in order to acknowledge what it actually is, to understand its issues, and to clarify which features must be included in or excluded from the models. Regarding reactive software systems, the analysis that precedes the modelling must identify every single entity within that system and within its environment - the Controller, all the Physical Entities, and all the Users - the roles they play, and how they do it. The following guidelines explain that process, defining the major concepts within it. Additionally, practical examples are provided, for a better understanding of the guidelines; the issues that shall deserve more attention from the modellers are pointed out, and solutions for those issues are detailed.

The whole process of analysis assumes that the design decisions are agreed among all the project stakeholders.

4.2.1 Identify the Physical Entities

The first thing to do when analysing a reactive software system is to identify its Physical Entities; these are divided in two categories: sensors and actuators.

Definition 5: Sensor
A sensor is a physical object that observes events from the system environment [49], and that warns the system when any relevant external event has occurred. Such warnings are presented to the system as stimuli and can be responded to.

To describe a sensor one must state which type of sensor is needed, what does the sensor do, and how does it do it. Since not all changes in the envi-

ronment are relevant to the system under development, it is also important to state which events must be reported to the system.

Definition 6: Actuator
An actuator is a physical object that can receive and interpret stimuli from the system environment, and that can also produce responses that affect that environment. Actuators can be *active*, if they have a behaviour of their own that is relevant to the system, or *passive*, if they do not have a behaviour of their own that is relevant to the system [9].

One must analyse the behaviour of actuators to ascertain which ones have active roles within the system and which ones are just passive actuators.

Example
In the Smart Library example, eight Physical Entities were identified: books, presence sensors, pressure sensors, lights, gates, displays, and personal local positioning devices (LPS - Local Positioning System).

In reactive software systems, sensors are always active entities, since their purpose is to acquire information from the environment and report it to the Controller. If the information a sensor captures is not relevant to the Controller, then that sensor has no use for that particular system. Thus, most probably, that sensor should not be included in the model as a Physical Entity. For the actuators, it does not work that way, as they may have active or passive roles within the system. None must be disregarded; passive actuators may be of equal or greater importance than any of the active ones. The question is: how to differ such roles? One can resort to English (or any other spoken language) to do so: for each actuator, the modellers should ask themselves whether they can or cannot describe any action, or interaction, of the actuator in the active voice; in the question, the subject of the sentence must be name of the entity being analysed, and the verb must be the action the modellers desire to reason about.

Example
In one of the possible scenarios of the Smart Library, it may be said "When the user approaches a book he is looking for, the lights on that book turn on". Although this can be said in the passive voice, e.g.: "the lights must be turned on", it is not a matter of choosing a syntax because it sounds better; what is important is that it is possible to describe the behaviour of a light

as an action that is actually performed by the light, and if it is possible to use the active voice to describe that action, then it can be assumed that that entity has at least one active role. On the other hand, if the modeller cannot describe any functionality using the active voice, then that entity is probably just a passive actuator. For example, in the Smart Library it cannot be said "The books do (something)" because the problem description does not assign anything for the books to do by themselves. Yet, it can be said "The books are handled by the users"; this means that the books are relevant to the system, because they are part of someone's action, but just by assuming a passive behaviour.

Once the Physical Entities are found and their roles unveiled, one can proceed to another guideline. Some Physical Entities may not be so evident in the beginning of the analysis, or just become necessary later; even the most experienced analysts may sometimes blurt out some information. It is not problematic, although it may delay the analysis and modelling processes. Forgetting some Physical Entities is not troublesome, if those which have been identified are sufficient to model some system functionality. Other Physical Entities can be added at a later stage. It may happen to be necessary to reconsider the dependencies between entities, and that can add some delays in the processes of analysis and modelling. But it is preferable to delay some proceedings, than having to re-do the entire analysis from scratch.

Identify the Physical Entities
Guideline Summary

Identify sensors and actuators. State which type of sensor is needed, what does the sensor do, and how does it do it. Distinguish active and passive actuators.

4.2.2 Identify the Users

The next thing to do is identifying all system users.

Definition 7: User
A user is any person who interacts directly with a system. There can be dif-

ferent kinds (or categories) of users, and they may exhibit different behaviour towards the system.

In order to avoid ambiguities at this step of the process, it is important to distinguish three particular system stakeholders: users, clients, and customers. In [39], this distinction is made clear:

> **Client:** *The person or organisation for whom the product is being built, usually responsible for paying for the development of the product.*
>
> **Customer:** *The person or organisation who will buy the product (note that the same person/organisation might play both the client, customer and sometimes user roles).*
>
> **User or End User:** *Someone who has some kind of direct interface with the product.*

This guideline concerns only the people referred as *End Users*, though, in many cases, there are people playing more than one role. Modellers must then become aware of this distinction, so they can recognize the issues that concern only the Users.

Users are characterized by having free-will, which makes discovering their actions a particularly challenging task. Users are, in the words of M. Jackson, *a biddable domain* [22], and thus, there are users on whose behaviour one can rely on, and there are others whose expected behaviour can never be ensured.

The process of discovering Users can be compared to discovering the actors of a system, in the use cases of UML.

Example

In the Smart Library example, only registered Users were considered; however, imagining other kinds of User, helps understanding the differences in behaviour. A librarian, for example, would not put books on the wrong shelves (at least, not on purpose), but the users of the library can do that. In this case, the librarian is biddable whilst the users are not. Such differences in behaviour have to be pointed out among different categories of users.

Identify the Users
Guideline Summary

Identify all the users. Distinguish between client, customer, *and* end user *to avoid ambiguities. Point out the reliable and the less reliable users.*

4.2.3 Identify functionality and structure

Both the Controller and the Users exchange data between them. Recalling fig. 4.1, the communications structure is defined by two interfaces, A and B. To describe each of these interfaces, one must identify: (1) which Physical Entities are connected to the Controller; and (2) which Physical Entities are connected with each category of Users, respectively.

To do so, one cannot avoid thinking about what the Controller and the Users have to do, how their actions affect the whole system, and how they contribute to the system operation. Three tasks (sub-guidelines) arise from this: (i) identifying the functions of the Controller; (ii) identifying the actions of each User; and (iii) identifying how such functions and actions engage with the Physical Entities.

Identify the functions of the Controller

The Controller is the *brain* of the system: it determines the actions the Physical Entities must perform, and by doing so, it creates a new behaviour that responds in a desired fashion to the system environment. The events that are external to the system are observed by the sensors and sent to the Controller as stimuli; the Controller then chooses an appropriate answer and enforces a predefined behaviour in the actuators. Such behaviour is called *desired behaviour*.

Definition 8: Desired Behaviour
A behaviour that is defined by the modellers and that reflects the actions they desire to see carried out by the system.

To describe the behaviour of the Controller it is necessary to identify which decisions, functions, and validations the system is supposed to make

or execute; this is equivalent to finding and describing UML *use cases* (at least some of them, because it may be the case that not all the *use cases* are related to the Controller). Finding out such information is finding out what is expected from the system to be designed; this is one of the most important steps in analysis, which makes it almost imperative that it results from a discussion between analysts, clients, and other stakeholders.

Example

In the Smart Library example, the main responsibilities of the Controller are:

- updating users' information in the displays as they enter and exit the library;
- identifying books and their positions by interpreting pressure sensor readings;
- showing users their books, by turning on/off lights on books and bookshelves;
- letting users enter and leave the library by opening and closing the gates;
- locating users' positions by interpreting readings from the presence sensors.

Identify the functions of the Controller

Guideline Summary

Identify which decisions, functions, and validations the system is supposed to make or execute.

Identify the actions of the Users

To identify the actions Users can perform within the system, it is necessary to learn how the Users interact with the Physical Entities. The modellers must acknowledge and understand what the Users are supposed to do with the system, and how they are supposed to behave (which can also be compared to identifying *use cases* in UML). Once again, this must result from a discussion between the project stakeholders.

Example

In the Smart Library, Users can:

- enter and exit the library through gates;
- handle books;
- request trajectories to books in their Local Positioning System devices.

Identify the actions of the Users

Guideline Summary

Identify what the users are supposed to do with the system, and how they are supposed to behave.

Identify the communications net

The two tasks above (*Identifying the functions of the Controller* and *Identifying the actions of the Users*) have almost determined, by themselves, the global structure of communication. Controller functions and Users' actions depend on Physical Entities to occur, and may affect those Physical Entities, when occurring; therefore, for each Physical Entity, it is necessary to determine whether there is a connection to the Controller, and whether there is a connection to the Users.

Not all the interactions between Physical Entities and Users have to play a contributory role to the system functioning; even though they may, in some way, be related to that system, they do not add functionality, nor impose restrictions on system operation. The modellers must find out which of those interactions are of interest to the future models, and which are not.

Example

In the Smart Library description, both the Controller and the Users interact with all the Physical Entities; however, not all of these interactions introduce by themselves information that is useful for the models, such as a User crossing a gate, or seeing the light of a book. In those situations, the system is not concerned about whether the User actually performed such actions or

not, but it is concerned with the respective consequences: crossing a gate means the user just entered or left the library, and seeing the light of a book, can denote the user have found the book he was looking for.

There may be some hierarchy within Physical Entities, but according to the architecture shown in fig. 4.1, on which these guidelines stand, that hierarchy must be abstracted from, by adding the Controller as an intermediary agent. Adopting (or not) such abstraction, remains as a decision for the modellers to make; the modeller shall decide whether the problem under study is large enough to make up for the changes that arise from such assumption. The level of modularity and configurability of the models remains as the modellers' decision. These guidelines aim to achieve a high degree of model configurability, and because of that, the view from the above architecture is adopted; but if configurability is not on the goals of the project, or does not have a great priority, then it may not be necessary to make that assumption.

Identify the communications net

Guideline Summary

Determine, for each Physical Entity, whether there is a connection to the Controller, and whether there is a connection to the Users. Highlight interactions of interest between Physical Entities and Users.

4.2.4 Identify the private phenomena of the Physical Entities

The private phenomena of an entity are its states and internal events (recall Definitions 2 and 3).

Regarding Physical Entities with active roles, one can start describing their behaviour by determining which are the states (and sub-states, if any exist) of those entities, and then identifying which possible actions can be performed to toggle among those states (or sub-states). Such actions comprise the entities' *assumed behaviour* (which comes opposed to *desired behaviour*).

Definition 9: Assumed Behaviour
A behaviour that reflects the actions performed by entities embedded in

the system under consideration. Such entities (like the Physical Entities) already exist in the real world, and exhibit a behaviour that is known by the modellers; thereby, it must not be modified, but acknowledged and used exactly as it may be observed in the real world objects that those entities represent.

Identifying the internal events strongly depends on the choices of the modellers and on the purpose of the models. The modellers may want their specifications to be so thorough that every atomic event must be discriminated, or they may want specifications that contain only the necessary events to achieve a functional model. Thereby, the identified internal events can be described in three different manners:

t∀ : one transition for all events, therefore, one transition for toggling between all states;

t∃ : one transition for each event, which means that the occurrence of that event results in a state that is not the current one (as the reader can confirm later in this chapter, this option must be used carefully, since it may hide information about the entity);

t∃s : one transition for each sub-event, which equals one transition for each pair of consecutive states (i.e., for each pair of states, where the second would be reached from the first, by the occurrence of a single event).

The option **t∃s** differs from **t∃** because it considers sub-events; their importance comes with the need of displaying all the truthful (and useful) information about a system. This is not always in the goals of a project, and therefore, sub-events are disregarded or maybe just unnoticed.

Definition 10: Sub-Event
A sub-event is an event split in two or more parts; although all parts lead to the same state, they start in different states.

The following example exposes the differences between the three options above.

Example
The lights of the books and bookshelves inside the library are *Turned On/Off* by the Controller; this can happen regardless the current state of the lights, either *On* or *Off*. So, how many events should be defined?

According to **t∀**, there would be only one event called, for example, *Toggle Light*, to do the work of the two events above.

According to **t∃**, there would be only the two events mentioned above (*Turn Lights On* and *Turn Lights Off*).

According to **t∃s**, there would be four possible events, because there are four pairs of consecutive states: (*On-On*, *On-Off*, *Off-On*, and *Off-Off*).

In the Smart Library, the project stakeholders agreed to the option **t∃**. Option **t∀** is poorly detailed and did not add much useful information to the models; for example, during a model simulation, it did not seem to clearly explain what was happening in the model. Option **t∃s** was too detailed, considering information that was not relevant within the goals of the models. The pairs *On-On* and *Off-Off* can be ignored when their absence does not affect the correct execution of the model.

The occurrence of an internal event may depend on internal and external conditions [49]. Examples of internal conditions may include time (i.e., an event that only occurs after a given amount of time) or the current state of the entity itself (i.e., an event that only happens when/if the entity finds itself in a specific state). Examples of external conditions may include commands sent by the Controller and actions performed by Users.

In sum, to describe the private phenomena of a Physical Entity one must identify its states, its internal events, and the restrictions on those internal events. Relevant attributes, like id, name, or others, must also be identified within this guideline.

Definition 11: Attribute
An attribute is a property of an entity [9]. Such property can assume a set of values (of a pre-defined range) that characterize the state of the respective entity.

As suggested in [8], attributes can be found by underlining key words and concepts, in the textual description of the problem being analysed. Yet, if no descriptions are available, writing one will undoubtedly help discovering these and other issues.

Identify the private phenomena of the Physical Entities
Guideline Summary

For each Physical Entity, identify the states, internal events, restrictions of internal events, and relevant attributes.

4.2.5 Identify phenomena shared between Physical Entities

The shared phenomena [12] are the states and events shared within the interfaces A and B (recall fig. 4.1). For each pair of communicating entities that were identified in the guideline 4.2.3, one must decide what type of data is shared between them. To help in this decision, one can face a state as data that is always available for others to read, and although it can be changed, it does not disappear; while an event can be seen as data that is only available for the first who *grabs it*; once it is consumed by some entity, it ceases to exist. Thus, states are permanent, but changeable, data, while events are ephemeral data.

Example

In the Smart Library example, the commands sent by the Controller to the lights are an example of shared phenomena. Each command from the Controller to the lights is a shared event, since each command can only be executed once.

In another example from the Smart Library, the Users can only enter and leave the library if there are any gates open, so the Users must be able to know, at any time, the state of the gates; therefore, the data about the gates is shared with the Users. It represents shared states, because that data is only *viewed* by Users and not actually *consumed* by them.

Identify phenomena shared between Physical Entities
Guideline Summary

Identify the type of data shared between communicating entities. Distinguish between permanent/changeable data and ephemeral data.

4.3 Modelling Guidelines

After studying the problem carefully, one can move on to drawing CPN diagrams. By using this language one guarantees the models can be executed and simulated.

The following guidelines explain how to use CPN models and *CPN Tools* to model reactive software systems.

4.3.1 Create pages

The first thing to do is to open *CPN Tools* in a new CPN model, and create a few new pages: one page for each sensor; one page for each active actuator; one page for each User; one page for the Controller; and, one page for the top-most module.

The top-most module is where the overall structure of the model is shown; it is addressed further in the description of the guidelines. Each page works as an individual module, therefore, by following this guideline, one guarantees model modularity. Other pages may be created to assist the management of the model (such as a page to initialize all the modules) but the modelling details of those pages are not addressed in these guidelines.

Create pages

Guideline Summary

Create one page for each entity and an extra page for the top-most module.

4.3.2 Draw the Physical Entities

In the previous analysis, active and passive actuators have been distinguished. This knowledge helps to decide which actuators shall be modelled as individual modules (the ones that have an active role) and which ones shall be modelled as simple tokens, or not be modelled at all (the ones that have a passive role). This guideline focuses on sensors and active actuators, and for each of these entities one must do the following five tasks: (i) draw states; (ii) draw internal events; (iii) set the data flowing direction; (iv) declare colour sets; and, (v) declare *CPN ML* primitives.

An example is presented later in this guideline, in order to better explain to the reader, the issues raised within the three first tasks.

Draw States

Every CPN module needs at least one place to hold the instances of the entity it represents. The modellers must decide which of the following options best suits their system under development, and their problem goals:

p∃ : one place for each possible state, so the location of a token is the current state (or sub-state) of that token; or

p∀ : one place for all the tokens, regardless of their current states.

The option **p∃** seems more intuitive and logical, if one desires to be able to rapidly understand the global state of a model, only by looking at it. In an analysis perspective this option seems better than the second; the states of the entities are dealt in a graphical form rather than being dealt within the colour sets, and manipulated with coding primitives. However, if being aware of the global state of a model is not a mandatory requirement, the option **p∀** can be chosen, and may be less burdensome, since it decreases the number of necessary states [11,23] (and therefore contributes to avoiding the *state-explosion problem*).

The modellers may opt to use one single *place* to hold all the tokens that represent an entity within the module of that entity, and deal with the states through the colour sets and coding primitives. In order to ease the references to such *place*, from this point on, is called *main place*.

Places may be either *state places* or *event places*. For example, a *main place* is always a *state place*, because it holds tokens with information about the state of an entity; this means that it always holds the same number of tokens that it initially possesses. Any transition connected to this kind of place can *peek* on the data inside it; the occurrence of such a transition can consume tokens, in order to change their values, but those tokens have to be restored right away. Consuming and restoring a token from a *state place* can be seen as a single, instantaneous action. In an *event place* (a place that holds tokens with information about events) the data within that place is stored by some transitions, and consumed by others (which are two different actions).

To start sketching actual models, the modellers must draw the state *places* (either one *main place*, or a place for each state of the entitty), with proper names and colour sets. The colour sets may be formally specified afterwards, but this way, the modellers can develop an early understanding of which data types will be needed later (just a few tasks after this one).

Draw States

Guideline Summary

For each Physical Entity, decide modelling either only one place, or one place for each possible state. Draw the place(s) with proper name(s) and colour set(s).

Draw internal events

If the decision about which internal events are needed has not yet been made (recall guideline 4.2.4) then here comes the time to make it. The modellers must draw a transition for every considered event and name each one properly.

Draw internal events

Guideline Summary

For each Physical Entity, decide modelling either only one transition, one transition for each event, or one transition for each sub-event. Draw the transition(s) and name it (them) properly.

Set the data flowing direction

The modellers can now draw the arcs between states and internal events, which will allow the data to flow along the model. Which data flows in each arc is the next question, and the answer is found in the next two tasks.

Based on what have been discussed so far about the modelling of events and the modelling of states, one can conclude that there are six possible ways of modelling the internal behaviour of a Physical Entity:

t∀ + p∀ : one transition for all events and one place for all states;

t∀ + p∃ : one transition for all events and one place for each state;

t∃ + p∀ : one transition for each event and one place for all states;

t∃ + p∃ : one transition for each event and one place for each state;

t∃s + p∀ : one transition for each sub-event and one place for all states;

t∃s + p∃ : one transition for each sub-event and one place for each state.

But are all these options really viable and trustworthy? The following examples illustrate the differences between some modelling approaches, and highlight a particularity of the option **t∃ + p∃** .

Example

Let us recall the lights example, which possible events and states were described in guideline 4.2.4.

The following images were not used in the Smart Library final model; they were only drawn to illustrate the differences between modelling approaches.

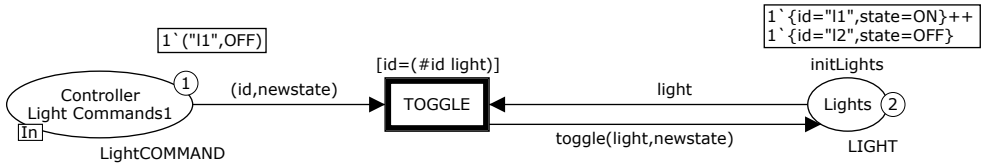


Figure 4.2: Modelling the *Lights* with one *main place* and only one transition

The example in fig. 4.2 is consistent with the option **t∀ + p∀**, where a single transition and a single place exhibit, in a simple, but poorly detailed, model, the complete behaviour of the lights; the real events are hidden by the expression in the outgoing arc from the transition, and the states can only be known by inspecting the colour set *LIGHT*. Whenever the Controller issues a light command, the transition *Toggle* can occur for the corresponding light; such occurrence can either change, or not change, the state of that light.

The example in fig. 4.3 is consistent with the option **t∃ + p∀**. In this model, the behaviour of the lights is detailed by two transitions. Whenever the Controller issues the command *(id,ON)*, the transition *Turn Light On*

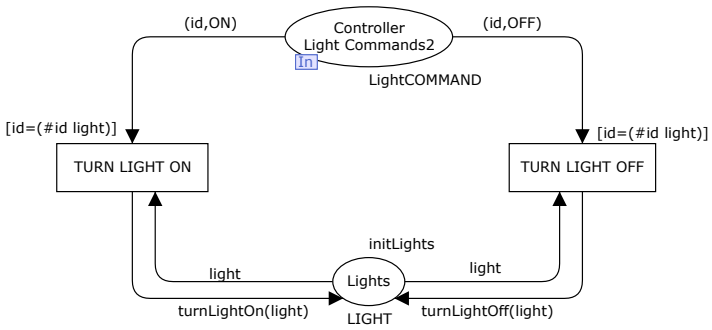


Figure 4.3: Modelling the *Lights* with one *main place* and one transition for each event

becomes enabled for the corresponding light; it works in a similar fashion for the *Turn Light Off* transition.

The example in fig. 4.4 is consistent with the option $\mathbf{t}\exists + \mathbf{p}\exists$; this approach can, and most probably will, introduce some errors, during the execution of the model. The step exhibited in that figure shows an enabled transition, because the conditions for that transition to occur are valid: the command *("l1",OFF)*, from the Controller to the *Lights*, tells to turn *off* the light *l1* and, at the moment, that light is *on*. But if that command was meant to the light *l2*, then that same transition (*Turn Light Off*) would not be enabled, because that light is already *off*. A command *("l2",OFF)* would not be consumed in this step; it could be consumed after, though, if the light *l2* was turned *on* by a new command; that would enable the *Turn Light Off* transition and the light *l2* could be turned off erroneously, because that was not the moment for that command to be consumed.

The example in fig. 4.5 is consistent with the option $\mathbf{t}\exists\mathbf{s} + \mathbf{p}\exists$, which is the most detailed approach for modelling the behaviour of the lights. In this model, the occurrence of a transition that is bound to a particular light, depends on the current state of that light (which did not happen with the remaining exemplified options).

There is a remarkable difference in the amount of code needed in each of the presented models: whereas in the first example, which uses a few amount of graphics, the code incorporates several decisions (Which light?, What is its current status?, What is the new state?) and actions (Change the state of that light to the new state), the last example, uses a greater

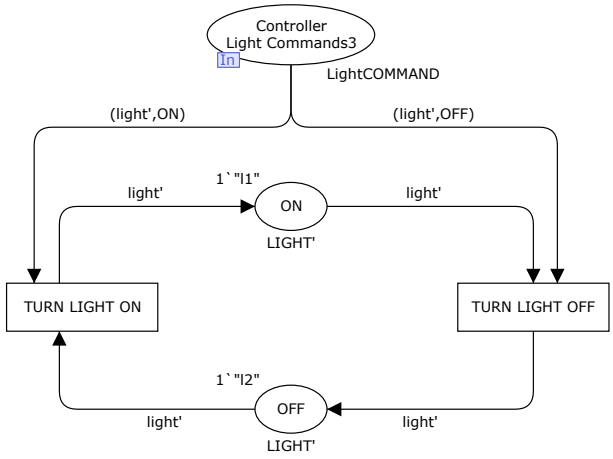


Figure 4.4: Modelling the *Lights* with one place for each state and one transition for each event

number of graphics, but it does not need any extra code (i.e., code beyond arc expressions) to describe all the behaviour of the lights.

Once the graphics for the Physical Entity have been outlined, then the modellers can proceed with some coding.

Set the data flowing direction

Guideline Summary

For each Physical Entity, draw the arcs between places and transitions.

Declare colour sets

Creating colour sets entails making more design decisions; each modelling approach may require specific data type structures, then it is important to spend some time thinking about what really is important to the problem and which are the best options.

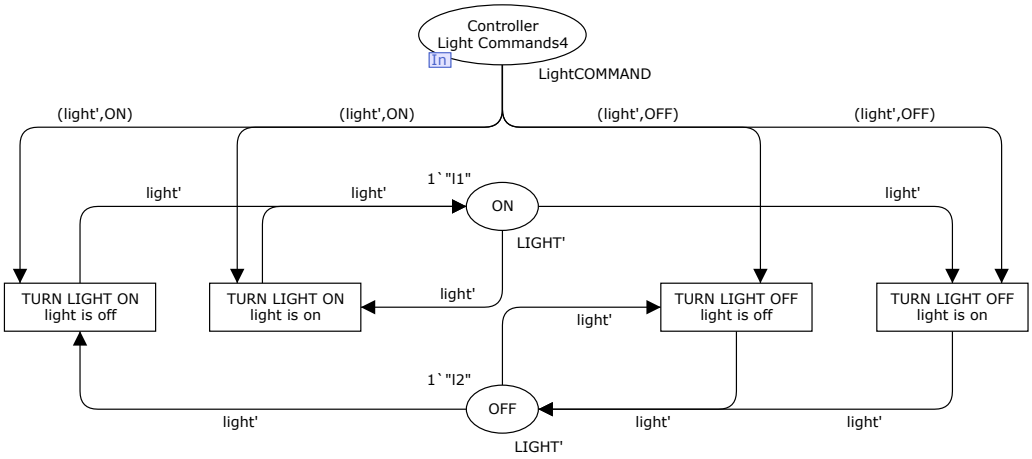


Figure 4.5: Modelling the *Lights* with one place for each state and one transition for each sub-event

In a first approach, the colour sets must be as simple as possible, only describing imperative attributes (which, depending on previous choices, may or may not include the states of the entity).

The decision of which colour set to use is not always trivial, and a particular problem comes with the use of the list colour set. In a coding perspective, lists are appealing, due to the many available primitives that help managing the data within the lists, for example, maintaining a particular order among tokens inside a list is easy, from a programmer's viewpoint. However, in a visual perspective, it is another matter; if one desires to achieve a model that can rapidly be read and understood, then lists are hardly the best choice. Lists are structured types, similar to arrays in general programming languages thereby, a list is represented by a single token, that may carry a lot of other tokens inside themselves. The graphics in CPN models show the number of tokens in each *place* at any moment, so, when using a list colour set, the number of tokens inside the *place* is always 1 (at least for each list inside the *place*); the tokens within the lists are not counted, whether the lists contain one, several, or no tokens at all. The only way to acquire such intel is by inspecting the lists, which may prove itself laborious when working with several lists, containing several tokens.

Example

The following code snippet shows three simple colour sets, and two compound ones, from the Smart Library example, declared in *CPN ML*:

```
1 colset ObjID = STRING;
  colset UID = STRING;
3 colset AreaID = with ENTRYAREA | A1 | A2 | B1 | B2 |
                       EXITAREA | OUTSIDE;
  colset USER = record id:UID *
5                      position:AreaID;
  colset ObjIDs = list ObjID;
```

Listing 4.1: A few colour sets used in the Smart Library example

The first and second lines declare the types of the Object and of the User's ids, respectively. The third colour set enumerates the different areas of the library; in this example, the library has seven areas where the Users are sensed by the presence sensors. The USER colour set declares a user as having an id and a position within the library. The last colour set declares a list of object ids; this is an example of how lists can be useful; this colour set is used, for example, in an attribute of the presence sensors, because each of these sensors needs to be aware of which Physical Entities are in its area of sensibility, so it can inform the Controller which objects are affected by its readings.

Declare colour sets

Guideline Summary

Declare the colour sets, only describing imperative attributes (in a first approach); other attributes can be added later.

Declare *CPN ML* primitives

The modellers can continue coding a little bit more, in order to create the functions and variables that are needed by the models (in the code segments of the transitions and in the arc expressions).

One must write the *CPN ML* primitives that validate the conditions imposed to the occurrence of each internal events (which means implementing the guards on transitions); and assign a behaviour to those events (i.e., implementing the code segments of each transition). A more experienced modeller may deal with time inscriptions right away, but these can also be dealt with later, once a first version of the model is functional and stable. CPN time inscriptions are not discussed in these guidelines.

Declare *CPN ML* primitives

Guideline Summary

Write CPN ML functions and variables for the code segments of each transition and for the arc expressions.

Draw interfaces for shared phenomena

The Physical Entities are not, by themselves, sufficient to complete the model. Once they assume graphical form in the CPN model and have their own behaviour, they must be prepared for establishing communication with other entities (the Controller and the Users). Thus, is it necessary to draw some interfaces to allow the exchange of phenomena between different entities.

Recalling what have been said in section 2.3.1, the CPN modelling language allows two representations of shared places: sockets and fusion places. A *fusion place* is like two (or more) places that were merged, and are accessed by two (or more) different entities, that share their contents, while a *port place* is a place that can communicate with another port place through a private communication channel (a socket). It can either be an unidirectional or a bidirectional socket. The doubt comes when one has to decide whether use fusion or port places, for each kind of data (states and events).

In *CPN Tools*, a socket can only be shared between two entities, and it is not possible to share port places. Since a *state place* must be accessible to several entities, port places are not suitable to depict such *places*. In contrast, fusion places are perfectly fit to share a *state place*; in *CPN Tools*, creating a fusion place from a common place, requires only tagging that common place with the desired fusion tag; all the *places* with a particular fusion tag become

representations of the same single place. Therefore, it is advisable the use of fusion places for shared states, and the use of port places for shared events.

To draw the interfaces for shared phenomena, the modellers must create the necessary fusion and port places, with the proper arcs, colour sets, and variables. Once again, the readers are advised to weight the pros and cons of using lists for such places. The lists do not favour the fast reading and comprehension of the model, but they are very useful to manage the tokens.

Example

The *Gate-Display* module shown in fig. 4.6 demonstrates the use of fusion and port places. The *Gate_Display Commands* place is a fusion place because it stores commands, which are events, sent by the Controller to this entity; yet, the place *Gates with Displays* is a port place, because it is a *state place* and the tokens within it are *consulted* by another module.

The two remaining places are also port places, but for another reason; these places contain data that can change for testing the model. Therefore, these places are shared with another module where all the places with configurable data are initialized for model simulation. That module is presented further in this chapter.

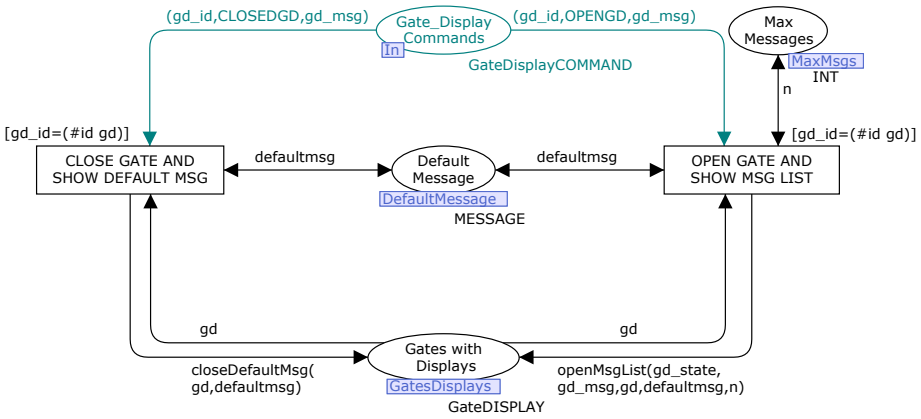


Figure 4.6: The *Gate-Display* module in the Smart Library example

The top-most module is built alongside the creation of these interfaces for shared phenomena; the modules of the Controller, of the Users, and of each Physical Entity are represented in the top-most module by substitution

transitions, which are connected by the corresponding interfaces for shared events. Because the top-most module does not have regular transitions (only substitution transitions) the arcs in this module do not exhibit expressions; the module only indicates which of the remaining modules has enabled transitions (if any does) in each step of a simulation.

Draw interfaces for shared phenomena

Guideline Summary

Draw places for interfaces; it is advisable the use of fusion places for shared states, and the use of port places for shared events. Draw the interfaces for shared-events in the top-most module, and create the corresponding substitution transitions.

4.3.3 Draw scenarios

Last but not least, one must create scenarios to depict the desired behaviours of both the Controller and the Users. Three tasks must be carried out to do so: (i) create initial values; (ii) create the desired behaviour of the Controller; and (iii) create the Users' desired behaviour.

Create initial values

To test a CPN model, it is necessary to create instances of Physical Entities, and initialize the proper *places* (the *main place* and/or the other state *places*) with those values. By doing so, the models become parameterisable.

Example

In order to expedite the Smart Library simulations, an auxiliary module was created to initialize the *main places* of the entities, the Users in the library, and other configurable data, with their respective values (because the module is too big, it is not shown at this point of the dissertation; however, it can be found in the appendix, in the section A.12, together with its thorough description).

The following code snippet exhibits the declarations of some of the variables:

```
val initUsers =
  1`{id="u1",position=OUTSIDE}++
  1`{id="u2",position=OUTSIDE}++
  1`{id="u3",position=A1}++
  1`{id="u4",position=ENTRYAREA}++
  1`{id="u5",position=EXITAREA};

val initLights =
  1`{id="l-a1",position=A1,state=OFF} ++
  1`{id="l-a2",position=A2,state=OFF}++
  1`{id="l-b1",position=B1,state=OFF}++
  1`{id="l-b2",position=B2,state=OFF};
```

Listing 4.2: Two variables used in the Smart Library example

Create initial values

Guideline Summary

Create instances of Physical Entities, and initialize the proper places.

Create the desired behaviour of the Controller

At this point of the process, the shared places must have already been drawn for the Controller and the Users' modules (recall guideline 4.3.2), which is a start; but, from now on, the modellers have to decide how to process the data that arrives in the input shared places, and to whom send the results of that processing, i.e., to which output shared places should tokens be added. Receiving, processing, and sending (or redirecting) data is, in short, the behaviour of the Controller. The question is: how to model the processing of data?

Entities like the Controller and the Users usually have many possible behaviours to be depicted; scenarios seem a suitable way of illustrating those behaviours. Creating scenarios requires creativity, and depends a lot on the nature of the problem under consideration, and how the modellers face it. In the words of Sinan Si Alhir:

> *It is important to stress that building and analyzing scenarios is a creative process of discovery* [2].

According to what must have been decided in the guideline 4.2.3, the Controller has a predefined set of tasks (the so called functionalities) that must be modelled. Each task can be carried out by a single and atomic (uninterruptible) action, or by a set of actions (an activity) that occur sequentially and that can be interrupted by the occurrence of events external to that activity. In addition to this, each task can be executed through different courses of action that may, or may not, produce the same results. The behaviour of the Controller must then be divided into scenarios that describe these tasks and their possible courses of action. Each scenario can be represented in individual modules, but they can also be included in the same module; since there are no evidences that one solution is better than the other, that remains as another choice for the modellers to make. The *divide to conquer* strategy may be a way of coping with complexity, but apart from that, the results obtained in model simulation and in formal analysis, do not exhibit differences between modelling scenarios separately, or modelling them together.

So, how does one create scenarios in *CPN Tools*? This can be compared to modelling with *sequence* or *collaboration diagrams* in UML, and similarly to those models, there are no rules to state exactly how to produce a scenario with CPN, but one can provide some ideas (or again, some guidelines) to do so. It is never too much to remind that guidelines propose a way of dealing with problems, and they do not give a mandatory solution, neither they try to overlook other solutions.

The first issue to consider in the modelling of scenarios refers to the representation of actions and activities. Because actions are non-inturreptible occurrences, then each action should be represented in a single transition; activities, on the other hand, should be represented in different, consecutive transitions, so they can be interrupted by external factors, if needed. As it was referred above, activities may sometimes be divided into alternative paths of execution, which may lead to the same result (the same final state) or to distinct ones (different final states). A particular strategy, presented and described in [14], may be used to model the divergent paths; at first, one must find the *variation points* (VPs) where a scenario splits into different courses of action. Those VPs are modelled by a place (let us call it place of divergency) with so many output transitions as the new paths that may be

followed. Those transitions are in concurrency with each other, so the occurrence of one of those transitions disables the others. Each transition leads to a different course of action, describing the entity's possible behaviours. Fig. 4.7 illustrates that strategy.

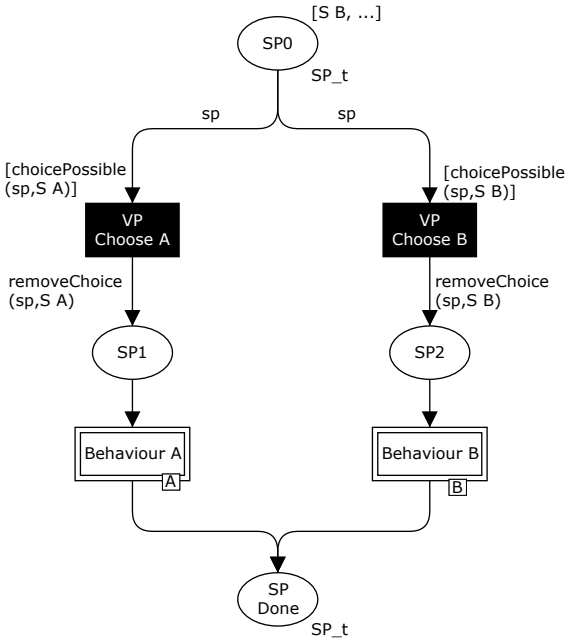


Figure 4.7: *Variation Points* leading to alternative behaviours (This image is a generic version of an example from [14] and is here in order to better explain the authors' strategy)

In fig. 4.7, the place of divergency is called *SP0*, and the black transitions are the VPs. Each VP leads to a possible behaviour that is referred by a substitution transition and described in a new module (of which no examples are shown). The token in *SP0* is a list of the choices to be followed along the execution of the model; in this example, the transition *VP Choose B* is the chosen one; other choices may appear after this one, if the module referred by the substitution transition *Behaviour B* also has alternative paths of execution.

Example

This scenario modelling strategy was applied in the Smart Library example,

but not strictly as it is described in [14]. For example, there is a scenario where the Controller must choose whether to issue a command for the Light module, or to issue a command for the GateAndDisplay module. The issuing of each command corresponds to a different behaviour, but as it can be seen in fig. 4.8, those behaviours are not described in different modules, because they are simple enough (just a single function) to be included in the same module.

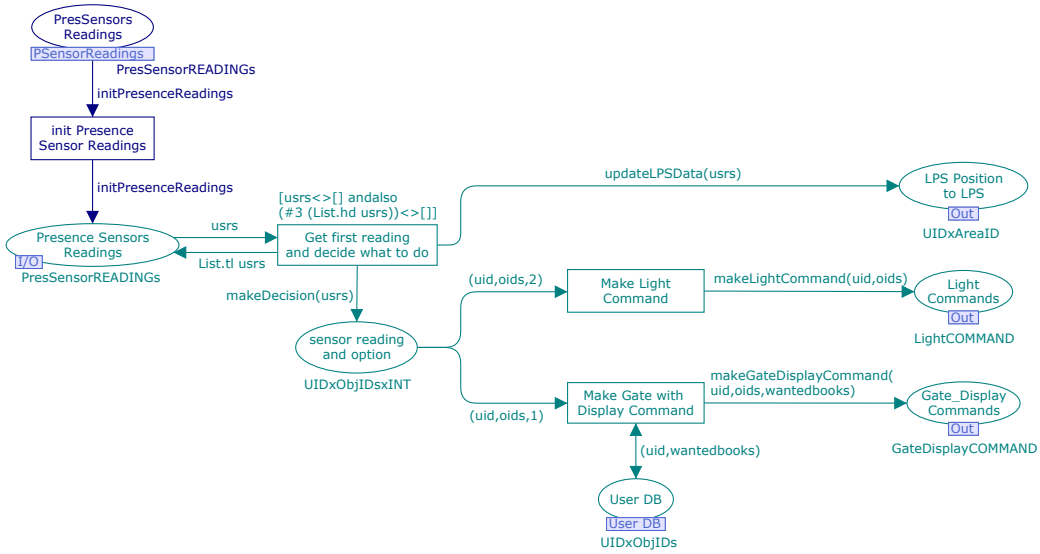


Figure 4.8: One scenario of the *Controller*

Another issue can be observed in the Controller module; the synchronization of actions. Because in the Smart Library example, gates and displays have to be synchronized, they are modelled as one single entity. In fact, when the Controller issues a command for the GateAndDisplay module, it should be issuing two commands, one for a Gate module, and another for a Display module, and those modules should process the commands at the same time. The CPN language provides primitives to model that situation, through transition synchronization; however, the *CPN Tools* does not support that technique.

There are three ways of working around the *CPN Tools* lack of support for transition synchronization; if one wishes to make sure that two different

actions occur at the same time, one can either:

- represent the two actions in the same transition;
- represent the two actions in separate transitions, and manually bind the next transition to be fired (which is only viable in interactive simulations);
- represent the two actions in separate transitions, and add a token as a resource that is shared with other transitions which are not synchronized with those two; once the token is consumed by the first of the two synchronized transitions, it is only restored when the second transition occurs, thus preventing other transitions of being enabled while the synchronized ones are still occurring.

Although viable, the last strategy must be used carefully because it may lead to deadlocks and overload of tokens, which may turn out to be cumbersome and uncessary management work. Although narrower, the other options are safer and easier to deal with.

In short, actions and activities may be modelled in the CPN modelling language as it follows:

- single action - single transition;
- synchronized actions (for automatic simulations) - single transition (safer choice) or competitive, sequential transitions;
- synchronized actions (for interactive simulations) - separate transitions;
- single activity, with only one course of action - sequential transitions without VPs;
- single activity, with more than one course of action - sequential transitions, with VPs.

One final thing must be pointed out about the modelling of scenarios. Scenarios are models prone to change; the conditions that support a given scenario may change rapidly, and when that happens, the scenarios need to be adapted according to the new conditions. If the problem being modelled is quite simple and is not subject to major changes, then the modellers can choose to create more graphical scenarios, which can be quickly read and

easily understood. On the other hand, if the problem is likely to change over time, requiring new scenarios to be considered or old ones to be disregarded, then the modellers should opt to draw less graphics and write more code, because code can be easily modified and is much more scalable. Thereby, it is advisable to build scenarios that are simple and easily configurable, and because of that, the modellers may sometimes have to choose code over graphics.

The use of scenarios, as they were described in these guidelines, benefits in two ways the modelling of reactive software systems with the CPN modelling language; it ensures the model can be easily configured, and

> *(...) the model reflects all the partial behaviours identified and discussed with the clients and users of the system under development.* [13]

Create the desired behaviour of the Controller

Guideline Summary

Draw actions and activities of the Controller. Each action should be represented in a single transition; activities, should be represented in different, consecutive transitions. Build scenarios that are simple and easily configurable (might have to choose code over graphics).

Create the Users' desired behaviour

What has been said in the previous task about modelling with scenarios, also applies to the Users' module.

Example

In the Smart Library example, the Users' module was modelled like is shown in fig. 4.9.

Users can perform four kinds of actions: move between areas of the library, request trajectories to their previously requested books in their LPS devices, and also pick up from and return books to bookshelves.

In the first action, *Users inside or near the Library* can *Move Between Areas* by choosing one path from the *Library Paths* place. This choice is

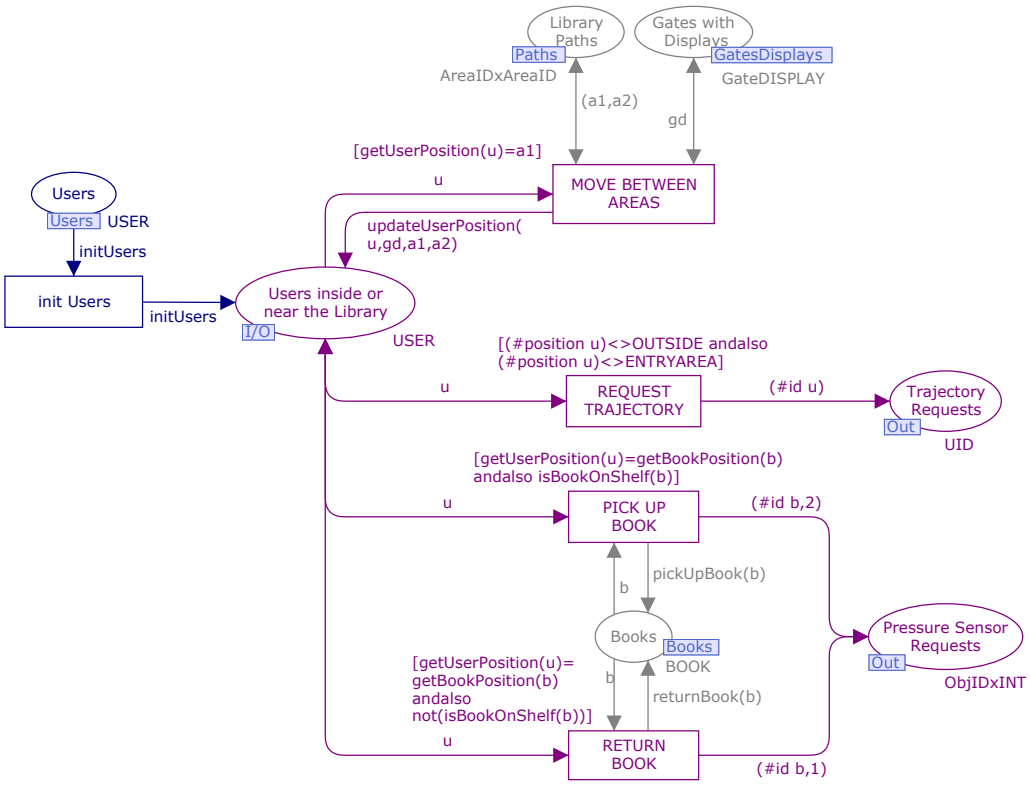


Figure 4.9: The User module in the Smart Library example

restricted by two conditions: a User can only move to another area, if there is a path from that User's position to the area he wants to go to, and that path is not blocked by a closed gate.

The second action is also restricted by two conditions, that make it impossible for a User to request a trajectory to a book while outside the library. If the User is inside the library (i.e., if the User has crossed one of the entry gates of the library) then the transition *Request Trajectory* can be executed.

The third and fourth actions refer to the handling of books, and they can only happen if the User is near the book (which is the first condition), and also if the book is on the shelf (in the *Pick Up Book* action) or off the shelf (in the *Return Book* action).

These actions could be modelled in different modules, which would demonstrate the scenarios independence from each other, and for complex scenarios

would help managing them; but that was not necessary because in the Smart Library example, the scenarios are quite simple (they can be described by a simple function), and separating them in different modules would not make it easier to manage them; on the contrary, it could be more cumbersome having to manage several pages to deal with simple scenarios.

Create the Users' desired behaviour

Guideline Summary

Draw actions and activities to be performed by Users . Each action should be represented in a single transition; activities, should be represented in different, consecutive transitions. Build scenarios that are simple and easily configurable (might have to choose code over graphics).

Summary

In this chapter, analysis and modelling guidelines are proposed, to model reactive software systems with the CPN modelling language.

The analysis guidelines start by identifying all the entities within the environment of a system (all the Physical Entities and all the Users) and then, identifying the main functions of the Controller and the actions that Users can perform. Afterwards, it must be identified how the entities are structured, i.e., how are they connected among themselves. The two last guidelines concern the identification of the internal behaviour and the external (or shared) behaviour of each Physical Entity.

The modelling guidelines start by explaining how to create a CPN model in *CPN Tools*. Then the Physical Entities are tackled, by means of modelling their states and events, designing their internal architecture, and declaring, in *CPN ML* their attributes and functions. Once the Physical Entities assume its structure and properties, the external architecture of the model can be dealt with, by creating the interfaces of communication for all the entities within the model. At last, scenarios must be created in order to depict the expected behaviours of both the Controller and the Users.

Chapter 5

Conclusions and Future Work

Prologue

This section gives a short overview of this dissertation, and explains its major contributions. Finally, some notes are given on how this work can be further developed.

5.1 Conclusions

This work presents an approach to modelling reactive software systems, more specifically controllers, by means of a set of guidelines that explain and exemplify their respective processes of analysis and modelling. By using CPNs as the modelling language, the results of this work contribute to creating formal and executable models, that can be used as specification of software systems.

This approach stands on an architecture capable of representing general controller systems, by differentiating three major components: Controller, Physical Entities, and Users [13]. This disjunction is based on the differences among the behaviours of each component, thus it is possible to apply appropriate modelling techniques for each individual behaviour. The Controller and the Users can be modelled through scenarios, which are formalisms able to represent desired behaviour, and the various Physical Entities can be modelled separatelly, representing their assumed behaviours.

Having a well-defined basis for constructing the models makes more explicit: (1) the overall system structure; (2) the various system components; and (3) the general communication network.

This architecture requires the existence of hierarchical modelling mechanisms, and the CPNs are a viable formalism in that way. Apart from hierarchy, the CPNs have a natural support to other characteristics that are thoroughly explored in the proposed guidelines.

The models built according to these guidelines benefit from modularity, parameterisation, configurability, and executability.

Model *modularity* allows to represent each component of a composite system individually, and then connect each other via communication interfaces, where they can share information. In the context of reactive software systems modelling, it must be possible to consider several types of user and several types of physical entities, and modularity makes it possible. The use of modules allows one to quickly include or exclude entities to a model, without having to change the other entities, whose behaviour remains the same.

Parameterization is used, rather than, once again, resorting to modules, to handle the instantiation of entities. Because module instantiation can be quite laborious in CPNs, it is preferable to represent different instances of an entity as token values. Storing parameters in tokens makes the models expandable, which leverages from easy management of resources. For example, the effort to model one user or one thousand users is similar, if users are represented by tokens; one either creates one token or one thousand tokens, which is easier than having to create that amount of module instances[1]. It would not be practicable having to create one thousand modules with exactly the same features and purpose. Apart from this, the use of parameters lays groundwork for configuration.

Configurability is achieved by the proposed guidelines by means of parameters and an initialization module, and by means of scenarios. The guidelines suggest the creation of an auxiliary module that sets up the configuration of the initialization module; when executed, that module hands out all the initial parameters to their respective entities. In a model with several modules it is important to gather in the same page all the configurable data, so no information is forgotten. The models also have configurable scenarios.

[1]Certainly, the initialization of one or one thousand tokens has its differences; initializing one thousand tokens requires either patience, or mastery of a language capable of generating the adequate values. But it still is simpler and quicker to create tokens, than to create modules.

Scenarios are a modelling technique for depicting several courses of action, thereby, it becomes possible to represent in the same model a great number os possible behaviours. The use of scenarios for modelling the Controller and the Users is suggested, due to the nature of their behaviours; these entities do not have the same well-established and well-known behaviour as Physical Entities do. The behaviour of the Controller depends on the purposes and goals of each problem, and therefore differs from one system to another; in addition to this, users are completely unpredictable and so, the representation of their behaviour requires discovering just a set of possible behaviours, that are of interest in the context of the problem. For those two components (Controller and Users), the description of their behaviour may denote multiple branches representing possible choices, one for each condition the component may be subjected to.

The development of models is a process of discovery [33]. Model *executability* supports experimentation [43], which is essential under such circumstances. This feature derives into model animation and simulation, which not only enable model testing, but also provides a good view of the model operation. The model testing, although shallow, is trustworthy and ensures a correct model operation; and model simulation helps in understanding the modelled system. Another benefit from executability and simulation, is that they make the models one step closer to being turned into prototypes, which in such an early stage of software development, can be a valuable asset for producing a useful and reliable system.

In addition to these features, any CPN model can be submitted to performance analysis and can be formally verified, which are desirable qualities in any software model. While *formal analysis* ensures the correctness of the model and allows one to study its dynamic properties, *performance analysis* enables to reason about the efficiency of the model, providing methods for finding out response times, delays, throughputs, and resource usage.

All the just described properties of CPNs make this modelling language an adequate tool for modelling reactive software systems, but also to similar systems, with non-trivial behaviours. The guidelines proposed in this dissertation are targeted at controllers, but other kinds of reactive software systems can also make use of their guidance such as real-time systems, embedded systems, and user interfaces, since the guidelines clearly distinguish each different step of the analysis and modelling processes.

Although CPNs are not object-oriented (OO), the guidelines are described in an OO perspective: identifying and modelling entities separately, each

with its own data and behaviour, resembles working with classes with their private attributes and private methods; describing and modelling interfaces for shared states and shared events resembles describing public attributes and public methods, respectively. Yet, many features of the OO paradigm are still missing such as polymorphism and inheritance (among others). Nevertheless, the approach pursued by the proposed guidelines tries to lessen the effects of the so-called "*gap between analysis and design models*" [8, 28], a common flaw in systems development, which hampers the evolution of software.

The CPN modelling language is undoubtedly adequate for modelling concurrency and parallelism, but it presents some drawbacks with respect to the modelling of scenarios. Scenarios are sequences of events happening in a predicted and controlled manner, with a well-defined structure; the CPNs do not allow to model such a structure as clearly as one could wish. UML, for example, provides a much better way of doing it, with Sequence Diagrams. Hence, an integration of CPNs and UML appears like something both languages would leverage from[2]. And since the guidelines aim the OO paradigm (which is on the foundations of UML), it would be possible to apply the guidelines to a modelling approach including both languages.

5.2 Future Work

These guidelines are not the one final solution for the modelling of reactive software systems; they are far from being perfect (after all, there is no such thing as *perfect guidelines*) and future changes can be done in order to better tackle the problem. As future work, it is intended to apply the guidelines to other demonstrative examples, including industry-level examples, in order to test and prove their suitability to different situations. As suggestions for future improvements, it is proposed addressing timed CPNs, systems with multiple controllers, and systems with multiple kinds of users.

It is also intended to study and analyse the feasibility of using a multi-language approach. An example would be joining CPNs and UML to create more structured scenarios.

This work can also proceed in the academic field, within the APEX

[2]In fact, in UML 1.5, the activity diagrams were restructured in order to adopt some concepts from the PNs [36]. This, however, does not make CPNs disposable, because the formal grounds and simulation techniques of CPNs are still better and more developed than those in UML.

project. Recalling the goals of the project, it seeks applying rapid prototyping for the study of human behaviour when facing software systems. The models for the Smart Library example are stable and complete enough for being tested and used by the project. They still need to be adapted to work with the APEX framework (which is a tool developed within the APEX project, that permits to create a virtual simulation of CPN models). Such adaption would require some changes in code, but the structure of the models is expected to remain untouched.

Also as future work, it would be important to analyse the developed models with the formal analysis techniques available in *CPN Tools*, in order to learn more about the correctness of the proposed solutions, and rectify those that might need to be improved.

Another interesting test to verify the applicability and the usefulness of the guidelines, would be to select a group of people (possibly students) with the same level of knowledge concerning the CPN modelling language, and ask them to model a practical example; half of the group would be using the guidelines, and the others would not. This would allow to acquire feedback about the usefulness of the guidelines, and about the issues (concerning analysis and modelling of reactive software systems) that should be more deeply addressed.

Bibliography

[1] Collins English Dictionary Complete & Unabridged 10th Edition. Guideline. Jan 2013.

[2] S.S. Alhir. *UML in a Nutshell: A Desktop Quick Reference*. O'Reilly Media, 1998.

[3] J.C.M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, May 2005.

[4] R. Bastide and P. Palanque. Petri Net Oobjects for the Design, Validation and Prototyping of User-Driven Interfaces. In *Human-Computer Interaction-INTERACT*, volume 90, pages 625–631, 1990.

[5] Conrad Bock. Three Kinds of Behavior Model. *Journal Of Object-Oriented Programming*, 12(4), July/August 1999. Available online at January 31, 2013.

[6] W. Brauer and W. Reisig. Carl Adam Petri and "Petri Nets". *Fundamental Concepts in Computer Science*, 3:129–139, 2009.

[7] F.P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE computer*, 20(4):10–19, 1987.

[8] P. Coad and E. Yourdon. *Object-Oriented Analysis (2. ed.)*. Yourdon Press computing series. Yourdon Press, 1990.

[9] B.P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2000.

[10] M. Elkoutbi and R.K. Keller. Modeling Interactive Systems with Hierarchical Colored Petri Nets. *Engineering*, 1997.

[11] J.M. Fernandes. *MIDAS: Metodologia Orientada ao Objecto para Desenvolvimento de Sistemas Embebidos*. PhD thesis, Dep. Informática, Universidade do Minho, February 2000.

[12] J.M. Fernandes, J.B. Jørgensen, S. Tjell, and J. Baek. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, APSEC '07, pages 294–301, Washington, DC, USA, 2007. IEEE Computer Society.

[13] J.M. Fernandes, S. Tjell, and J.B. Jørgensen. Requirements Engineering for Reactive Systems with Coloured Petri Nets: the Gas Pump Controller Example. *CPN 2007 - 8ht Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 207–222, 2007.

[14] J.M. Fernandes, S. Tjell, J.B. Jørgensen, and O. Ribeiro. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net. pages 44–53, 2007.

[15] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Menlo Park, CA, 1996.

[16] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[18] C. Girault and R. Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[19] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–74, 1987.

[20] D. Harel and A. Pnueli. *On the Development of Reactive Systems*. Weizmann Institute of Science, Department of Computer Science, 1985.

[21] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1998.

[22] M. Jackson. Problem Analysis and Structure. *Nato Science Series Sub Series III Computer and Systems Sciences*, 180:3–20, 2001.

[23] K. Jensen. An Introduction to the Practical Use of Coloured Petri Nets. *Lectures on Petri Nets II: Applications*, pages 237–292, 1998.

[24] K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[25] K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *Software Tools for Technology Transfer*, 2(2):98–132, 2007.

[26] J.B. Jørgensen and J. Baek. Coloured Petri Nets in Development of a Pervasive Health Care System. In WilM.P. Aalst and Eike Best, editors, *Applications and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 256–275. Springer Berlin Heidelberg, 2003.

[27] J.B. Jørgensen, S. Tjell, and J.M. Fernandes. Formal Requirements Modelling with Executable Use Cases and Coloured Petri Nets. *Innovations in Systems and Software Engineering*, 5(1):13–25, 2009.

[28] H. Kaindl. Difficulties in the Transition from OO Analysis to Design. *Software, IEEE*, 16(5):94–102, sep/oct 1999.

[29] M. Köhler. *Object Petri Nets: Definitions, Properties, and Related Models*. Univ., Bibliothek des Fachbereichs Informatik, 2003.

[30] C.A. Lakos, C.D. Keen, and TAS Hobart. Simulation with Object-Oriented Petri Nets. In *Australian Software Engineering Conference, Sydney*, 1991.

[31] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, 2002.

[32] C. Maier and D. Moldt. Object Coloured Petri Nets - A Formal Technique for Object Oriented Modelling. *Concurrent Object-Oriented Programming and Petri Nets*, pages 406–427, 2001.

[33] J. Marincic, A.H. Mader, and R.J. Wieringa. Explaining Embedded Software Modelling Decisions. In *IEEE CS International Conference on Software Science, Technology, and Engineering, SWSTE 2012, Herzlia, Israel*, pages 80–89, USA, 2012. IEEE Computer Society.

[34] J. Niu, J. Zou, and A. Ren. OOPN: Object-oriented Petri Nets and Its Integrated Development Environment. In *Proceedings of the Software Engineering and Applications, SEA*, 2003.

[35] C. Nyce and API CPCU. Predictive Analytics White Paper. *American Institute for CPCU. Insurance Institute of America*, pages 9–10, 2007.

[36] Omg. OMG Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1). Technical Report OMG Document Number: formal/2011-08-06, Object Management Group, August 2011.

[37] R.G. Pettit and H. Gomaa. Modeling State-Dependent Objects using Colored Petri Nets. In *CPN 01 Workshop on Modeling of Objects, Components, and Agents*, 2001.

[38] D. Ram and M. Rajasree. Enabling Design Evolution in Software through Pattern Oriented Approach. In Dimitri Konstantas, Michel Lonard, Yves Pigneur, and Shusma Patel, editors, *Object-Oriented Information Systems*, volume 2817 of *Lecture Notes in Computer Science*, pages 179–190. Springer Berlin / Heidelberg, 2003.

[39] J. Robertson and S. Robertson. Volere. Requirements Specification Template. Edition 6.0. Technical report, Atlantic Systems Guild, 1998.

[40] J. Rumbaugh. *OMT Insights: Perspectives on Modeling from the Journal of Object-Oriented Programming*. Press Syndicate of the University of Cambridge, 1996.

[41] J.A. Saldhana and S.M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. *Electrical Engineering*, pages 1–26, 2000.

[42] R. Schuette and T. Rotthowe. The Guidelines of Modeling An Approach to Enhance the Quality in Information Models. In Tok-Wang Ling, Sudha Ram, and Mong Lee, editors, *Conceptual Modeling ER*

98, volume 1507 of *Lecture Notes in Computer Science*, pages 240–254. Springer Berlin Heidelberg, 1998.

[43] B. Selic. The Pragmatics of Model-Driven Development. *Software, IEEE*, 20(5):19–25, 2003.

[44] F. Shull, I. Rus, and V. Basili. How Perspective-Based Reading Can Improve Requirements Inspections. *Computer*, 33(7):73–79, 2000.

[45] J.L. Silva, O.R. Ribeiro, J.M. Fernandes., J.C. Campos, and M.D. Harrison. The APEX Framework: Prototyping of Ubiquitous Environments Based on Petri Nets. In *Proceedings of the Third International Conference on Human-centred Software Engineering*, HCSE'10, pages 6–21, Berlin, Heidelberg, 2010. Springer-Verlag.

[46] I. Sommerville. *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[47] W.M.P. van der Aalst. Three Good rReasons for Using a Petri-Net-Based Workflow Management System. *Information and Process Integration in Enterprises*, pages 161–182, 1998.

[48] W.M.P. van der Aalst, A.H.M Hofstede, and M. Weske. Business Process Management: A Survey. In WilM.P. Aalst and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2003.

[49] R.J. Wieringa. *Design methods for reactive systems - Yourdon, Statemate, and the UML*. Morgan Kaufmann, 2003.

[50] E.S.K. Yu. Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235. IEEE, 1997.

Appendix A

Models for the Smart Library

This appendix presents and explains the model developed for the Smart Library example. The model is split in the following modules:

Top-Most The module that shows the overall architecture of the model.

Lights The module that shows the assumed behaviour of the lights.

Gates with Displays The module that shows the assumed behaviour of the gates and the displays.

Presence Sensors The module that shows the assumed behaviour of the presence sensors.

Pressure Sensors The module that shows the assumed behaviour of the pressure sensors.

LPS The module that shows the assumed behaviour of the LPS devices.

Users The module that shows the desired behaviour of the users.

Controller - scenario: Presence Sensor Readings The module that shows the desired behaviour of the controller related to the processing of the readings from the presence sensors.

Controller - scenario: Pressure Sensor Readings The module that shows the desired behaviour of the controller related to the processing of the readings from the pressure sensors.

Controller - scenario: LPS Requests The module that shows the desired behaviour of the controller related to the processing of requests for destination, from the LPS devices.

Controller - scenario: Trajectory Requests The module that shows the desired behaviour of the controller related to the processing of requests for trajectory, from the LPS devices.

Init The module that sets up all the configurable data, and initializes the model.

Before proceeding to the presentation of the modules, it is important to explain the difference between the representation of the Users' scenarios and those of the Controller; why are the scenarios of the Controller split in various modules, while the scenarios of the Users are not?

In a first approach, the scenarios of the controller were modelled together, since they are not too complex and they can fit distinctly in the same *CPN Tools* page. However, in order to make them easier to explaining and more comprehensible, it was decided to present them in separate modules. The separation of scenarios in pages does not affect any functionality nor any basic structure of the model; it merely changes the visual appearance of the model. The same would not happen with the Users' module, since all scenarios in that module need the same port place. The only way to perform such division of the scenarios would be to change that port place into a fusion set, but that would go against what is suggested by the guidelines proposed in this dissertation (recall 4.3.2, where the guidelines suggest the use of port places for the shared events, and of fusion sets for shared states.)

The reader might notice different colours (not to be confused with colour sets) in the design of the model:

- *Black* graphics concern what belongs to that module;
- *Green* graphics concern what belongs to the module of the Controller and to its communication interfaces;
- *Pink* graphics concern what belongs to the module of the Users and to its communication interfaces;
- *Blue* graphics concern initialization data;

- *Grey* graphics concern global variables that do not need to actually belong to a module, but that are used by one or many modules.

The modules presented next do not exhibit any simulation feedback, to avoid an overload of information in the images.

A.1 Top-Most module

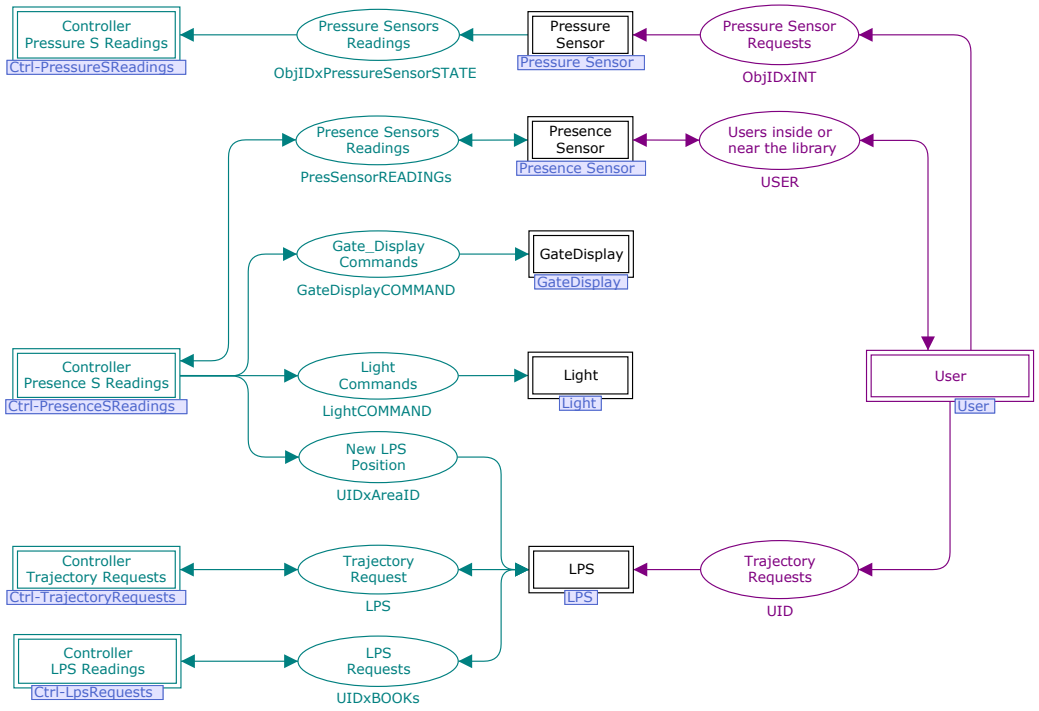


Figure A.1: The *Top-Most* module

The Top-Most module in fig. A.1 shows the overall architecture of the model. The structural resemblance to the architecture presented in fig. 4.1 is evident: the substitution transitions on the left are all related to the Controller and to its scenarios; the substitution transitions in the middle represent all the Physical Entities; and the substitution transition on the right represents the Users. The places between substitution transitions represent the interfaces of communication, in a similar fashion to the one shown in fig. 4.1.

Because this is the primary module that only shows the hierarchy of the model, the arcs do not have any expressions. During simulation, this module only shows which modules are enabled in each step, rather than showing the flowing of data.

A.2 Lights module

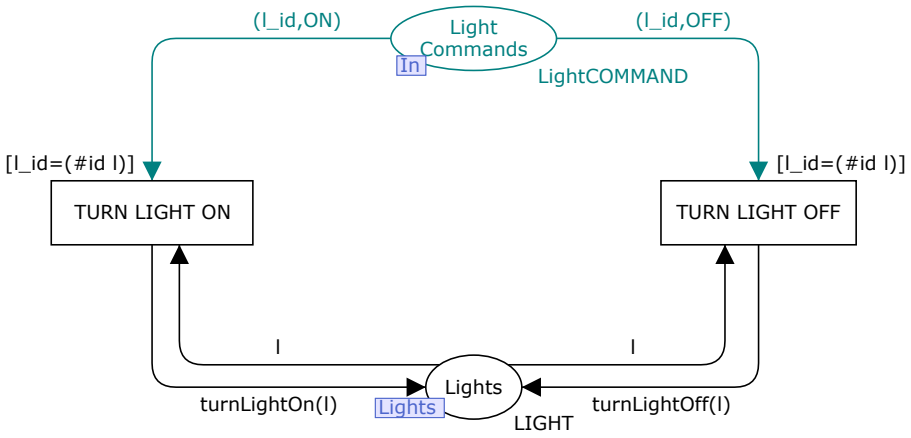


Figure A.2: The *Lights* module

Fig. A.2 represents the module of the Lights and it is composed by:

- one main place called *Lights*, where the instances of lights are stored;
- one place called *Light Commands*, that makes the interface with the Controller and that stores the commands sent by it;
- two transitions (*Turn Light On/Off*) that represent the assumed behaviour of the Lights.

A command issued by the Controller to this module carries the id of the light that it concerns, and the state to which that light must be changed. The *Turn Light On* transition will be enabled for any light *l* with *id=l_id*, if an *(l_id,ON)* command is issued. The *Turn Light Off* transition occurs in a similar fashion, with the command *(l_id,OFF)*.

A.3 Gates with Displays module

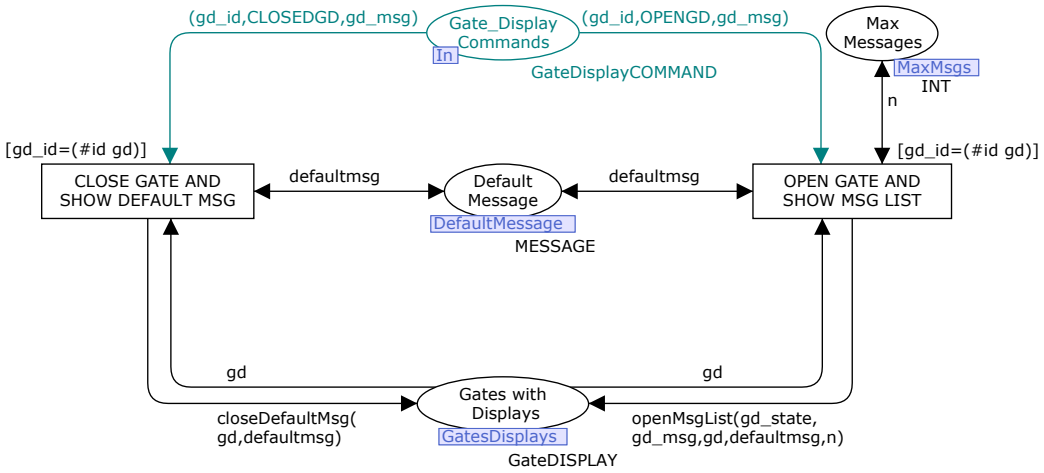


Figure A.3: The *Gates with Display* module

This module concerns the behaviour of Gates and the behaviour of Displays; because these behaviours must be synchronized, the two Physical Entities were modelled together, as an entity with a behaviour that corresponds to the occurrence in parallel of the two behaviours.

Fig. A.3 is composed by:

- one main place called *Gates with Displays*, where the instances of gates with displays are stored;
- one place called *Gate_Display Commands*, that makes the interface with the Controller and that stores the commands sent by it;
- two transitions (*Close Gate and Show Default Msg* and *Open Gate and Show Msg List*) that represent the assumed behaviour of the gates with displays;
- two other places that store data concerning the displays. The place *Default Message* contains a string representing a default message that appears in the display when nobody is near the gate with display. The place *Max Messages* contains an integer the maximum number of messages to be shown, at the same time, in the display.

A command issued by the Controller to this module, carries the id of the gate with display that it concerns, and the state to which the gate with display must be changed. The *Close Gate and Show Default Msg* transition will be enabled for any gate with display *gd* with *id=gd_id*, if an *(gd_id,CLOSEDGD,gd_msg)* command is issued. The *Open Gate and Show Msg List* transition occurs in a similar fashion.

A.4 Presence Sensor module

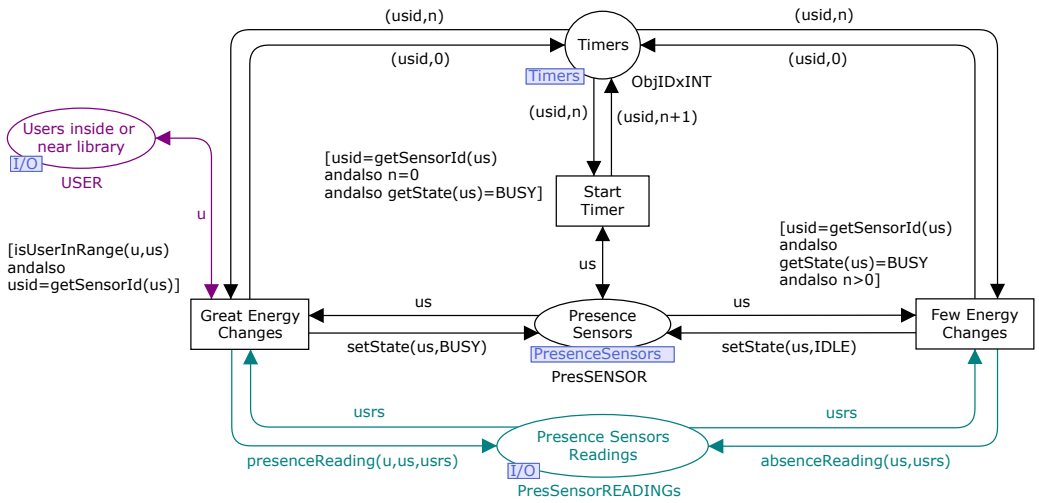


Figure A.4: The *Presence Sensors* module

Fig. A.4 represents the module of the Presence Sensors, and it is composed by:

- one main place called *Presence Sensors*, where the instances of presence sensors are stored;
- one place called *Presence Sensors Readings*, that makes the interface with the Controller and that stores the readings from these sensors to be forwarded to the Controller;
- one place called *Users inside or near the library*, that makes the interface with the Users and that stores instances of Users;
- two transitions (*Great Energy Changes* and *Few Energy Changes*) that represent the assumed behaviour of the presence sensors;
- one place *Timers*, that stores a counter (a timer) for each instance of presence sensor;
- one transition *Start Timer*, for starting the timer of a sensor, whenever that sensor is *Busy* (the sensor is either *Busy* or *Idle*).

When a user u approaches a sensor *us* (*us* stands for *user sensor* because this sensor is for detecting users, only), then the *Great Energy Changes* transition becomes enabled for that sensor. When there no users near the sensor, and the corresponding timer has been started, then the *Few Energy Changes* becomes enabled for that sensor. The occurrence of any of these transitions puts a token in the place *Presence Sensors Readings* informing the Controller that the user u is now, or is no longer, near the sensor *us*.

All timers have the initial value 0 (zero). Whenever a presence sensor is *Busy*, the *Start Timer* transition can occur, increasing the value of the corresponding timer. That value is reset by the occurrence of the *Great Energy Changes* transition, indicating that the sensor is now, or continues, *Busy*, and by the *Few Energy Changes* transition, indicating that the sensor is now *Idle*.

A.5 Pressure Sensor module

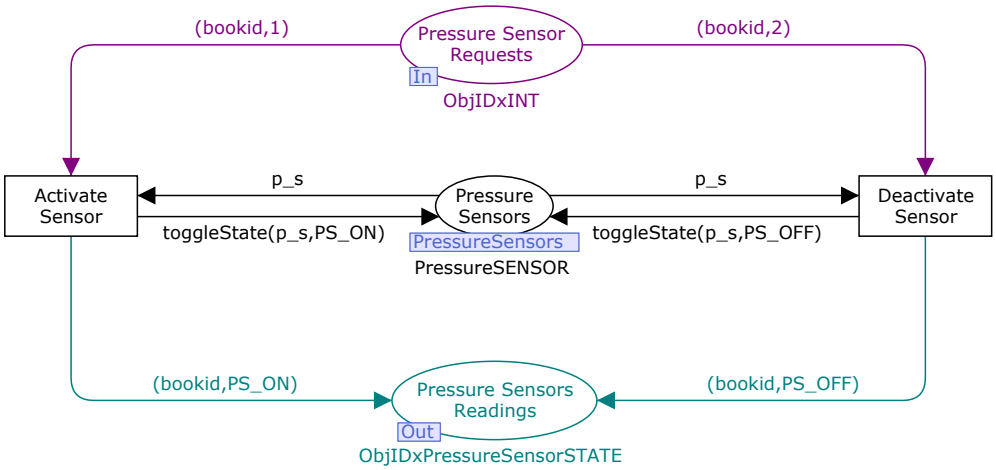


Figure A.5: The *Pressure Sensors* module

Fig. A.5 represents the module of the Pressure Sensors, and it is composed by:

- one main place called *Pressure Sensors*, where the instances of pressure sensors are stored;
- one place called *Pressure Sensors Readings*, that makes the interface with the Controller and that stores the readings from these sensors to be forwarded to the Controller;
- one place called *Pressure Sensor Requests*, that makes the interface with the Users and that stores requests for toggling the state of a pressure sensor;
- two transitions (*Activate Sensor* and *Deactivate Sensor*) that represent the assumed behaviour of the pressure sensors.

In this module, it is assumed that any book has a one-to-one relationship with one pressure sensor; for each book, that is one pressure sensor, and that sensor recognizes only that book.

When someone returns a book to its position on a bookshelf (represented by a request *(bookid,1)*), then the *Activate Sensor* becomes enabled for the sensor corresponding to that book; similarly, when someone picks up a book from its position on a bookshelf (represented by a request *(bookid,2)*), then the *Deactivate Sensor* becomes enabled for the corresponding sensor.

The occurrence of any of these transitions puts a token in the place *Pressure Sensors Readings* informing the Controller that the sensor related to the book with *id=bookid* is now on the state *PS_ON* or *PS_OFF*, respectively.

A.6 LPS module

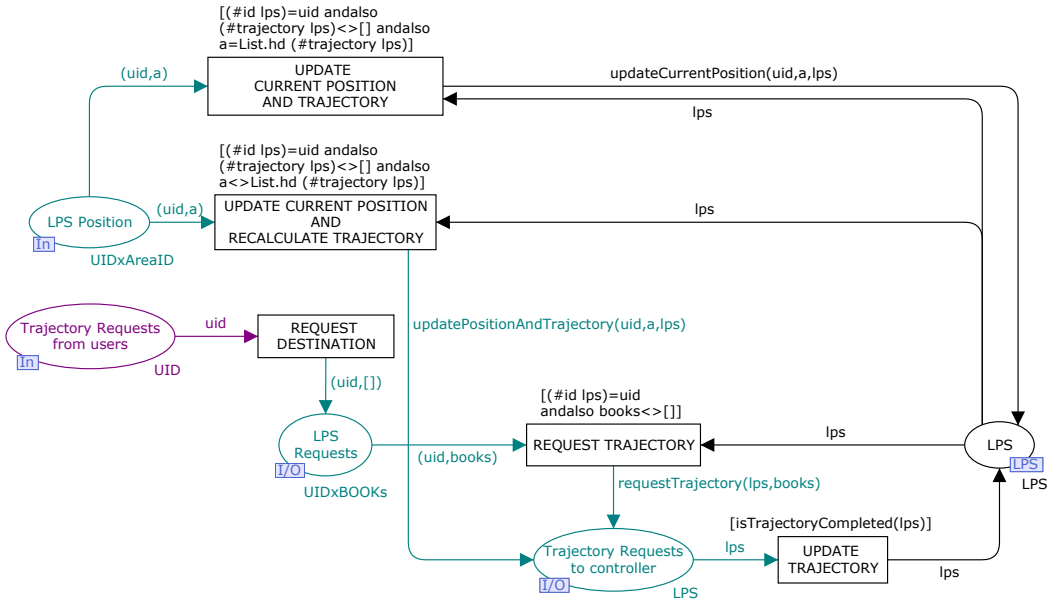


Figure A.6: The *LPS* module

Fig. A.6 represents the module of the LPS devices, and it is composed by:

- one main place called *LPS*, where the instances of LPS devices are stored;
- three places (*LPS Position*, *LPS Requests*, and *Trajectory Requests to controller*) that make the interface with the Controller;
- one place called *Trajectory Requests from users*, that makes the interface with the Users, and that stores requests from these, to acquire trajectories from their current position to the location of one of their requested books;
- five transitions (*Update Current Position and Trajectory*, *Update Current Position and Recalculate Trajectory*, *Request Destination*, *Request Trajectory*, and *Update Trajectory*) that represent the assumed behaviour of the LPS devices.

In this module, it is assumed that any LPS device has a one-to-one relationship with the Users; for each user, that is only one device, and that device carries only the data corresponding to that user. It is also assumed that the system knows the books each user has requested; the process of requiring a book is not addressed in the model of the Smart Library.

Whenever there are tokens in the place *Trajectory Requests from users*, the transition *Request Destination* can occur, placing a token *(uid,[])* in the *LPS Requests* place. This last place stores requests (to be processed by the Controller) to acquire the list of a user's requested books. Once that list is known by the LPS device, and if it is not empty, than the *Request Trajectory* transition can occur for that device, placing a request for a new trajectory in the place *Trajectory Requests to controller*; those requests are fulfilled by the Controller, which calculates the trajectory from a user's position, to one of his/hers requested books, and then sends that trajectory back to the place *Trajectory Requests to controller*.

The place *LPS Position* stores the current position of users (sent to the Controller by a Presence Sensor, and then sent to this module by the Controller). The transitions that follow this place can only occur if a trajectory has already been calculated for the LPS devices corresponding to the users' ids in that place.

The first transition, *Update Current Position and Trajectory*, becomes enabled only for users that follow the trajectory that was suggested to them; when it occurs, it updates the corresponding LPS device position.

The second transition, *Update Current Position and Recalculate Trajectory*, becomes enabled only for users who do not follow their suggested trajectory; when it occurs, it issues a request for recalculating the trajectory, by adding a token in the place *Trajectory Requests to controller*.

A.7 Users module

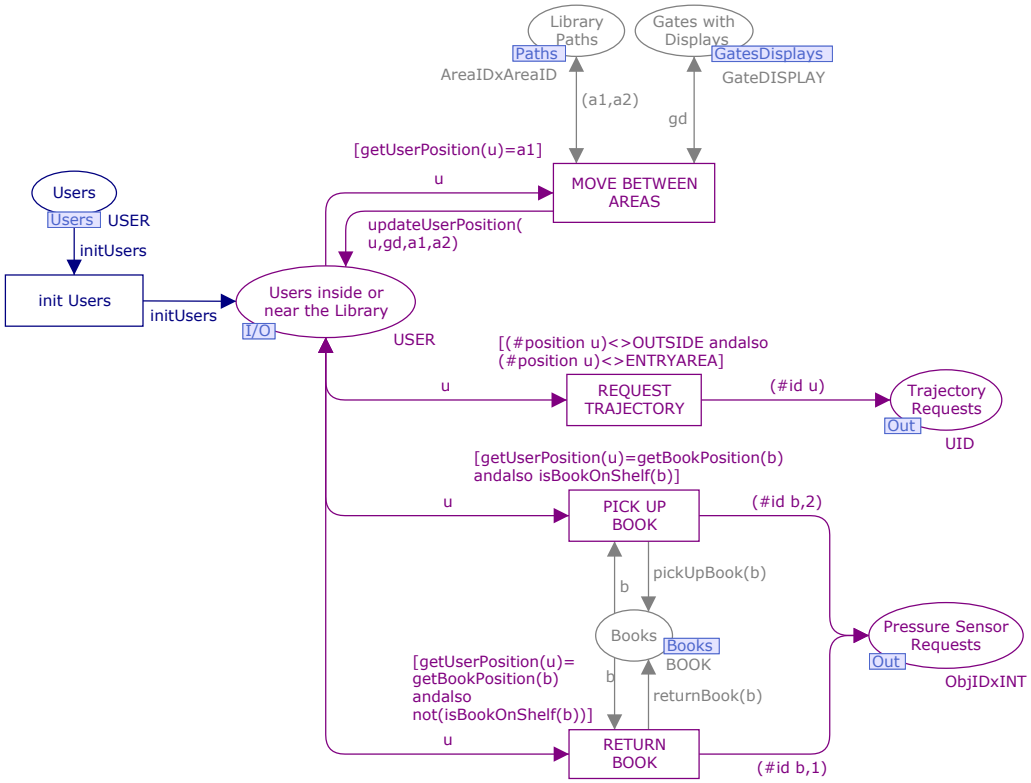


Figure A.7: The *Users* module

Fig. A.7 represents the module of the Users, and it is composed by:

- one place *Users* and one transition *init Users* that serve only to initialize the place *Users inside or near the Library*;
- one place *Users inside or near the Library*, where the instances of users are stored;
- four transitions (*Move Between Areas*, *Request Trajectory*, *Pick Up Book*, and *Return Book*) that represent the users' desired behaviour;
- one place *Trajectory Requests*, where the requests for acquiring trajectories to requested books are stored;

- one place *Pressure Sensor Requests*, that stores information about the moving a book from/to its location on the bookshelf, i.e., whether a book was picked up from, or returned to its bookshelf;

- three places (*Library Paths*, *Gates with Displays*, and *Books*) that hold the global data (global variables) which are used within this module.

This module considers only registered users.

All the four, pink, transitions refer to actions that users may perform. The occurrence of the transition *Move Between Areas*, changes the current position of the corresponding user, according to the data stored in the places *Library Paths* and *Gates with Displays*.

The occurrence of the transition *Request Trajectory* puts a token in the place *Trajectory Requests*, with the id of the user who is making the request.

The occurrence of the transitions *Pick Up Book* and *Return Book*, updates the state of the corresponding book (in the place *Books*), and adds a token to the place *Pressure Sensor Requests*, with the id of that book, and a number indicating how it was moved (*1* if it was returned, and *2* if it was picked up).

A.8 Controller scenario - Presence Sensor Readings module

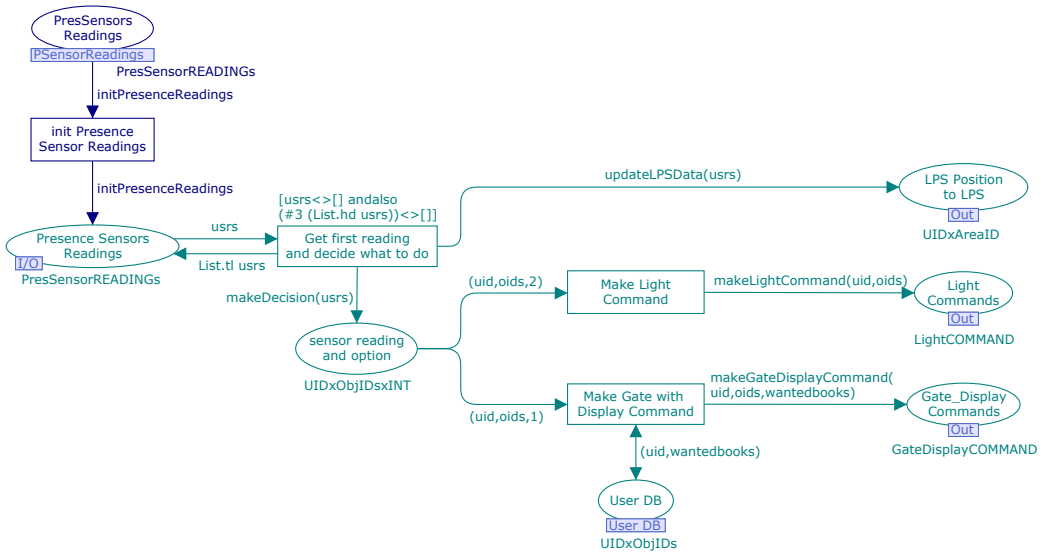


Figure A.8: The scenario of the *Controller* that relates to the processing of data received from presence sensors

Fig. A.8 represents the scenario of the *Controller* that relates to the processing of data received from presence sensors. This module is composed by:

- one place *PresSensors Readings* and one transition *init Presence Sensor Readings*, that serve only to initialize the place *Presence Sensors Readings*;

- one place *Presence Sensors Readings*, that stores the data coming from presence sensors;

- three places (*LPS Position to LPS*, *Light Commands*, and *Gate_Display Commands*) that make the interfaces with the modules LPS, Lights, and Gates with Displays, respectively;

- one place *User DB*, where the Controller keeps its information about the registered users;
- three transitions (*Get first reading and decide what to do*, *Make Light Command* and *Make Gate with Display Command*) and one place (*sensor reading and option*), that represent the desired behaviour of the Controller in this scenario.

In this module it is assumed that the objects inside an area that is sensed by a presence sensor, can only be of one kind, out of two: Lights, or Gates with Displays.

Every reading from the presence sensors, leads to the occurrence of two, out of three, possible events. The first event sends the user's position, acquired from that reading, to the LPS module; this event is performed by the transition *Get first reading and decide what to do*, which will always be enabled for any reading from the presence sensors.

The remaining events, *Make Light Command* and *Make Gate with Display Command*, reflect two possible courses of action in this scenario: depending on which objects are sensed by the presence sensor, the Controller either issues a command for the Lights, or a command for the Gates with Displays.

A.9 Controller scenario - Pressure Sensor Readings module

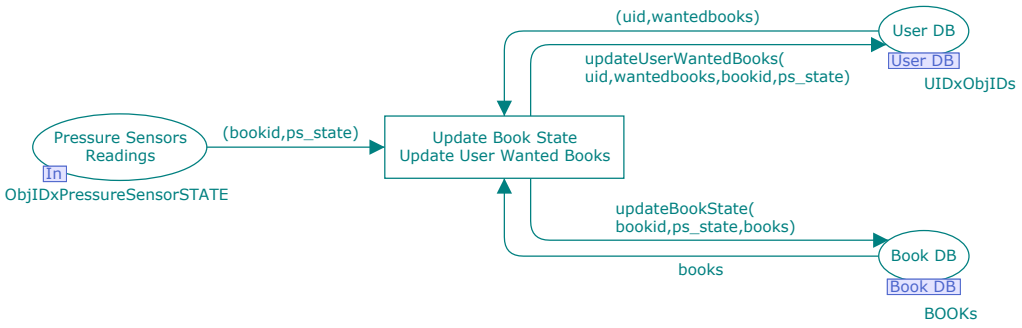


Figure A.9: The scenario of the *Controller* that relates to the processing of data received from pressure sensors

Fig. A.9 represents the scenario of the *Controller* that relates to the processing of data received from pressure sensors. This module is composed by:

- one place *Pressure Sensors Readings*, that stores the data coming from pressure sensors;
- two places (*User DB* and *Book DB*), where the Controller keeps its information about the registered users, and about the books of the library;
- one transition *Update Book State - Update User Wanted Books*, that represents the desired behaviour of the Controller, in this scenario.

Whenever a book is removed from, or returned to, its position on the bookshelf, the corresponding pressure sensor is deactivated or activated, respectively. When that happens the Controller is informed, in order to be able to update its own information concerning the state of the book, and concerning the user that handled that book. If the book was picked up from its position by a user that was looking for it, then the list of requested books from that user must be updated, reflecting this occurrence.

A.10 Controller scenario - Destination Requests module

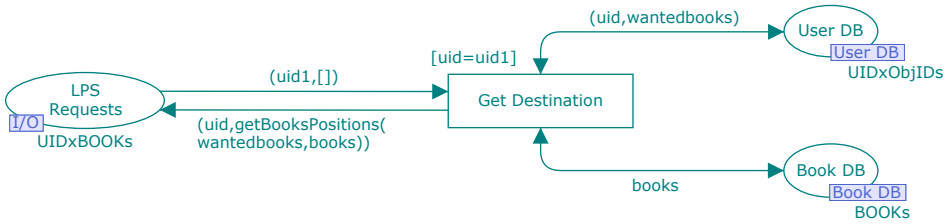


Figure A.10: The scenario of the *Controller* that relates to the processing of data received from LPS devices, for acquiring new destinations

Fig. A.10 represents the scenario of the *Controller* that relates to the processing of requests from LPS devices, for acquiring a new destination. This module is composed by:

- one place *LPS Requests*, that stores the requests for new destination coming from LPS devices;
- two places (*User DB* and *Book DB*), where the Controller keeps its information about the registered users, and about the books of the library;
- one transition *Get Destination*, that represents the desired behaviour of the Controller, in this scenario.

When a user requests a trajectory to its own LPS device, that Physical Entity asks the Controller about two things: the first (*Request Destination*) is explained in this scenario, and the second (Request *Trajectory*) is explained in the next scenario.

This scenario works in a similar fashion to the previous one A.9: the Controller *consults* its own information about the users and the books of the library, to discover the location of one of the books requested by a user.

A.11 Controller scenario - Trajectory Requests module

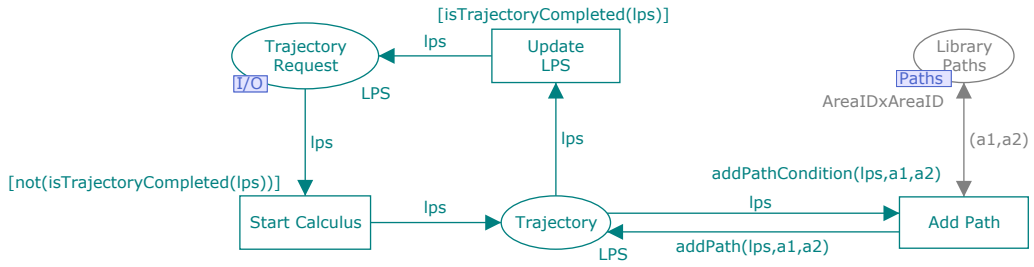


Figure A.11: The scenario of the *Controller* that relates to the processing of data received from LPS devices, for acquiring new trajectories

Fig. A.11 represents the scenario of the *Controller* that relates to the processing of requests from LPS devices, for acquiring a new trajectory. This module is composed by:

- one place *Trajectory Requests*, that stores the requests for new trajectory coming from LPS devices;
- three transitions (*Start Calculus*, *Update LPS*, and *Add Path*) and one place (*Trajectory*), that represent the desired behaviour of the Controller, in this scenario;
- one place *Library Paths*, that the Controller *consults* in order to choose one of the possible paths inside the library.

Whenever a token is sent to the place *Trajectory Requests*, exhibiting a trajectory that is not yet completed (i.e., the trajectory does not end at final destination, which is the location of a particular book), the *Start Calculus* transition can occur. When it does, the *Add Path* transition becomes enabled for the corresponding LPS device (the one that requested this calculation) and keeps adding possible paths to the trajectory of that device, until one of those paths leads to the final destination. The choice of the path to add next, does not follow any optimization algorithm, it only ensures that in every two subsequent paths of the trajectory, the second path starts where the first ends.

A.12 Init module

The last module is shown in fig. A.12.

The places on the right side of the transition *Initialize Simulation* are the places for initializing the other modules of the Smart Library; each group in an horizontal direction refers to a different module (except for the those in the first horizontal group, that refer to global variables).

Reading from top to bottom it can be seen:

- *Library Paths* and *Books*, which refer the global variables;
- *Users*, representing the User's module;
- *Presence Sensors Readings*, *User DB*, and *Book DB*, which refer to the Controller module;
- *Presence Sensors* and *Timers*, which refer to the Presence Sensors module;
- *Gates with Displays*, *Default Message*, and *Max Messages*, which refer to the Gates with Displays module;
- *Lights*, which refers to the Lights module;
- *LPS*, which refers to the LPS module;
- *Pressure Sensors*, which refers to the Pressure Sensors module.

The remaining places, *Init* and *Run*, show (by having, or not having, one token) whether this module has already, or not yet, been started.

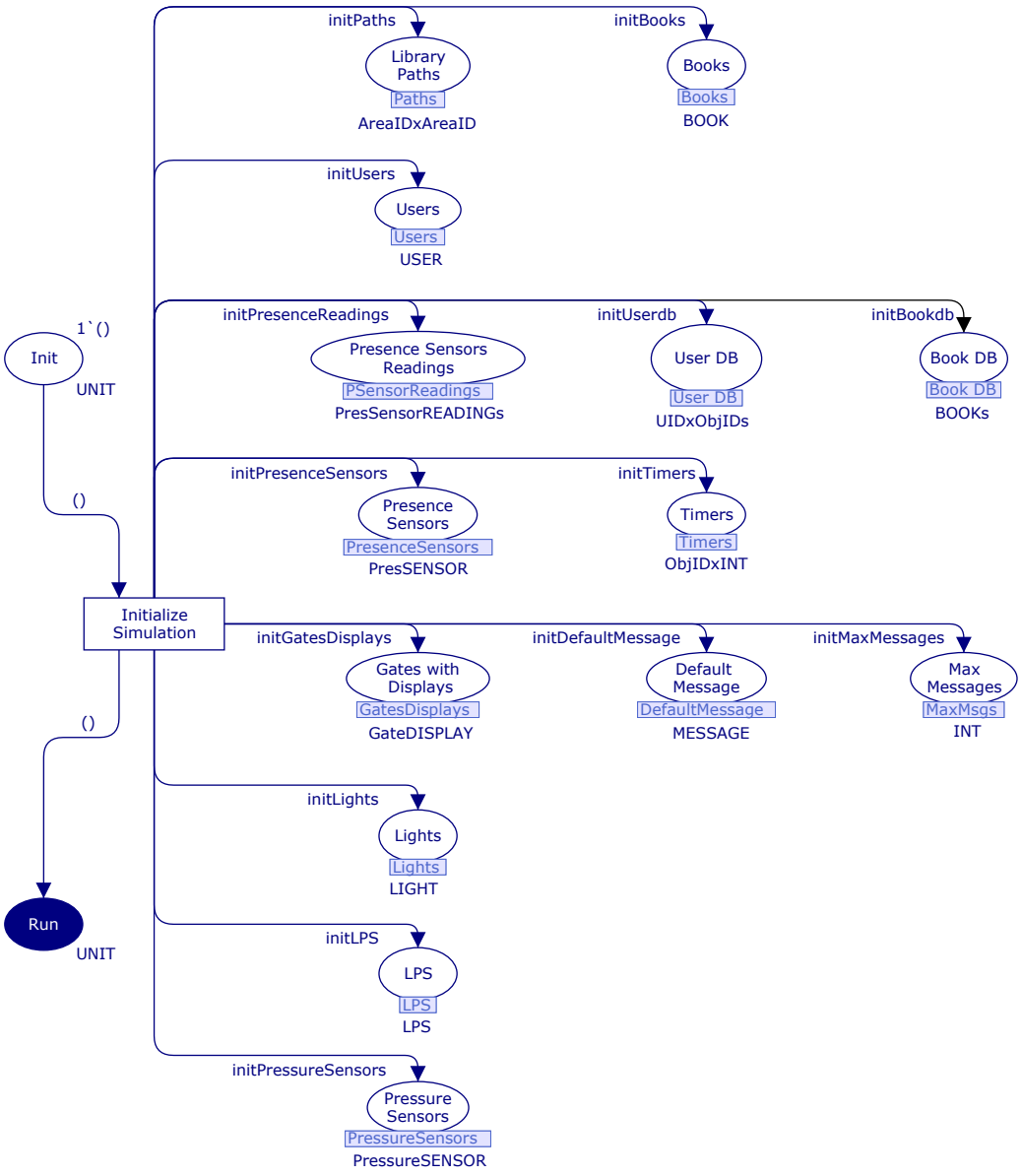


Figure A.12: The *Init* module