# "Mapping Live Sequence Chart into Coloured Petri Nets for Analysis and Verification of Embedded Systems"

Por

**Leonardo Amorim de Barros**

**Dissertação de Mestrado**

RECIFE, MARÇO/2006

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Leonardo Amorim de Barros

# "Mapping Live Sequence Chart into Coloured Petri Nets for Analysis and Verification of Embedded Systems"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: Dr. Paulo Romero Martins Maciel

RECIFE, MARÇO/2006

Barros, Leonardo Amorim de

Mapping Live Sequence Chart into Coloured Petri Nets for Analysis and Verification of Embedded Systems / Leonardo Amorim de Barros. – Recife : O Autor, 2006.

xii, 134 folhas : il., fig.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2006.

Inclui bibliografia e anexos.

1. Ciência da computação – Sistemas digitais. 2. Sistemas embarcados – Análise e verificação de propriedades. 3. Mapeamento de Live Sequence Chart(LSC) para Coloured Petri Nets(CPN) – Verificação de propriedades. I. Título.

```
004.414.23      CDU (2.ed.)          UFPE
004.21          CDD (22.ed.)     BC2006-363
```

# Acknowledgements

I would like to thank Dr. Paulo Maciel for his patience, friendship and confidence in my work, as well as, for his guidance through this project.

Thanks to my friend Meuse Nougueira, who contribute with valuable informations in a case study presented in this work.

Thanks to TechnoOK Corporation, represented by Frederico Braga, for having allowed that one of its products could be used as a case study.

Thanks to Dr. Ricardo Massa and Cleyton Moura, who helped me in the development of the engine presented in this work.

Thanks to my friends Dr. Raimundo Barreto, Eduardo Almeida and Eduardo Valentim for their contributions in this work.

Thanks to C.E.S.A.R Company for having stimulated and helped me in the attainment of the master degree.

Last, but not least, I would like to extend a thanks to my parents, Joao de Barros and Carmem Lucia, to my sister, Liliane Amorim, to my friend, Bernardo Caldas and to my girlfriend, Luciana, for their support and patience.

## Abstract

Nowadays, embedded systems are present in almost any human interacting environment and activities. The crescent adoption of embedded-system-controlled machines is direct related to the decreasing costs of such systems.

Due to the cost and the complexity of an embedded system architecture, it is essential high-level system design tools and methods, where functional and architectural description validation and verification might be carried out.

A more recent way to specify requirements, which is popular in the realm of object-oriented systems, is to use Message Sequence Charts (MSC). Both MSC and UML's sequence diagrams specify scenarios as sequences of message interactions between object instances. Scenarios capture the desired relationships among the processes, tasks, or object instances. Such models are applied for describing what the system should execute, but they do not allow designers specifying what must not be carried out (anti-scenarios).

Live Sequence Chart (LSC) is an MSC extension that allows the specification of anti-scenarios as well as activities that must occur. LSC fills out the gaps of the previous models, distinguishing things that can happen of things that must happen, through the use of some types of diagrams.

Nowadays, no tool and method is available for LSC properties' verification. Therefore, this works proposes a PN model for LSC language as a mean for allowing verification and analysis of system's properties.

**Keywords:** Petri Net, Coloured Petri Net, Modeling, Specification Languages, LSC, Properties Analysis.

# Resumo

Atualmente, os sistemas embarcados estão presentes em quase todas as atividades e ambientes do homem. A crecente adoção dos sistemas embarcados está diretamente relacionado com a queda do custo de tais sistemas.

Devido ao custo e complexidade da arquitetura de um sistema embarcado, é essencial ferramentas e métodos de alto nível que permitam a validação e verificação dos requisitos funcionais e arquiteturais do sistema.

A forma mais recente de especificar requisitos, que é popular no âmbito dos sistemas orientados a objetos, é o Message Sequence Chart (MSC) ou diagrama de sequências (UML). Ambos especificam os cenários com uma sequência de interações de mensagens entre instâncias de objetos. Os cenários capturam a relação desejada entre processos, tarefas e instâncias de objetos. Tais modelos são utilizados para descrever o que o sistema pode fazer, mas não permitem especificar o que deve ocorrer, assim como também não permite a modelagem dos anti-cenários.

A linguagem Live Sequence Chart (LSC) é uma extensão da MSC que permite especificar anti-cenários, assim como também permite modelar o que deve ocorrer. A linguagem LSC preenche a lacuna dos modelos anteriores, distinguindo as coisas que podem ocorrer das coisas que devem ocorrer através da utilização de alguns tipos de diagrama.

Atualmente, não existe uma ferramenta ou método que permita verificar propriedade para um cenário LSC. Então, este trabalho propõe um modelo de Rede de Petri para a linguagem LSC, através do qual propriedades do cenário LSC podem ser analisadas e verificadas.

**Palavras chaves:** Redes de Petri, Redes de Petri Coloridas, Modelagem, Linguagem de Especificação, LSC, Análise e Verificação de Propriedades.

# Contents

# List of Figures

# List of Symbols

| | |
|---|---|
| **CPN** | Coloured Petri Net |
| **CPN ML** | Coloured Petri Net ML Language |
| **ASKCTL** | CPN Model Checking Query Language |
| **LSC** | Live Sequence Chart |
| **MSC** | Message Sequence Chart |
| **GUI** | Graphical User Interface |
| **PN** | Petri Net |
| **XML** | eXtensible Markup Language |
| **UML** | Unified Modeling Language |
| **XUML** | Executable Unified Modeling Language |
| **RTOS** | Real Time Operating System |
| **SA** | Structured Analysis |
| **SD** | Structured Design |
| **OOAD** | Object-Oriented Analysis and Design |
| **RISC** | Reduced Instruction Set Computer |
| **VLIW** | Very Long Instruction Word |
| **OMG** | Object Management Group |
| **DSP** | Digital Signal Processor |
| **JDOM** | Java API for XML Processing |

# Chapter 1

# Introduction

*This chapter presents an introduction to embedded systems, highlighting the importance of the specification in the initial phase of a embedded system project, where several specification models are presented revealing their advantages and disadvantages, as well as presenting some methods that allows to verify if a specification is according to the functional and nonfunctional requirements imposed by specified system.*

## 1.1 Context

Embedded systems are present in practically all human activities, and due to low technological costs, they tend to increase their presence. Examples of such systems are cellular telephones with camera and calendar, cars and buses system controls, portable computers, microwave ovens with an intelligent control of temperature, washing machines and other appliances.

An embedded system project is a complex activity, because it involves concepts such as portability, energy consumption constraints, performance, low memory availability, safety and reliability.

Embedded systems projects face several challenges, because there is a vast design space to be explored. The hardware architecture of an embedded system may contain one or more processors, memories, interfaces for peripherals and dedicated blocks. These components are linked by a communication structure that can vary from a bus to a complex net [37]. The processors can be of several types (RISC, VLIW, DSP) according to the application. In the case of systems containing programmable components, the software application may be composed of multiple processes, distributed among different processors and communicating through varied mechanisms. A real time operating system (RTOS) offering services as communication and processes stagger may be necessary [13]. Besides the precious time that can be spent with a systematic exploration of this project space, it should be considered the necessary time for designing and validating all dedicated components of the system, processors, hardware blocks, software routines, RTOS, as well as the time for validating the whole system.

Another problem of an embedded system project is the cost. The project of an embedded system of great complexity is quite expensive for a company, involving different teams (digital hardware, analogical hardware, software, test) and usually demands specialized tools often of very high costs. The production cost of integrated systems in a tablet is high, so the companies are pushed to implement components that have a high production volume, in order to reduce the production costs.

Starting from a high level specification, the design space should be exploited for possible architectural solutions, taking into account the impact of different hardware and software solutions. After or while defining the architecture, the communication should be considered for the synthesis integrating the hardware components [36].

Due to the complexity of embedded system architectures, containing multiple hardware and software components, sophisticated communication structure, the variety of possible solutions, performance, energy consumption constraints, correctness and robustness, it is essential the adoption of tools to automate design phases and supporting designers in decision making.

An embedded system project usually begins with a specification of each desired functionality, done through a language or an appropriate formalism. Ideally, this specification should have a high abstraction level, which is independent of implementation, hardware components or software. This specification should be preferably executable, for validation ends.

Design methodologies should provide means for functional and architectural validation. Improving system reliability can be carried out by simulation or through formal analysis/verification that is quite attractive because they spare exhausting simulations.

The initial system specification is usually a functional description, in which no structural information or architecture dependent features is considered. This description should be neutral in relation to possible implementations of software functions or hardware platforms, and usually does not contain detailed information on how to implement the timing requirements. The system is described as a group of functions (tasks or objects, depending on the adopted language), that communicate through high-level communication primitives, for instance in the form of messages or services requests. Each request may transport several items of data simultaneously. This abstraction level allows the validation of the functional specification of the system and serves as input for the architectural exploration process.

Over the last decades, the main approaches to high-level system modeling have been structured-analysis/structured-design (SA/SD) and object-oriented analysis and design (OOAD). Both approaches have yielded visual formalisms for capturing the various parts of a system model. The linking of structure and behavior is crucial and by no means a straightforward issue. In SA/SD, for example, each system function or activity is associated with a state machine or a state chart [22] that describes its behavior. In OOAD, as Unified Modeling Language (UML) [40, 18] and the XUML [23], each class is associated with a state chart, which describes the behavior of every instance object.

When developing a complex system, it is very important to be able to test and debug the model before investing extensively in implementation [21].

Requirements are the basis for testing and debugging models. They constitute the constraints, desires, and hopes we entertain concerning the system under development. We want to make sure, both during development and when we feel development is over, that the system does, or will do, what we intend or hope for it to do.

A more recent way of specifying requirements, which is popular in the realm of object-oriented systems, is to use Message Sequence Charts (MSCs) [14]. The International Telecommunication Union (ITU) adopted this visual language as a standard.

Both MSCs and UML's sequence diagrams specify scenarios as sequences of message interactions between object instances. In the early stages of system development, engineers typically come up with use cases [27] and then specify the scenarios that instantiate them. Scenarios capture the desired relationships among the processes, tasks, or object instances. The modeler uses MSCs to specify the scenarios, that the final system hopefully will satisfy and support, and these scenarios are instantiations of the more abstract and generic use cases.

As a requirement language, all known versions of MSCs, including the ITU standard and the sequence diagrams adopted in the UML, are weak in expressive power. Their semantics are little more than a set of simple constraints on the partial order of possible events in a system execution. Nothing can be said in MSCs about what the system will actually do when it runs. These diagrams can state what might possibly occur, not what must occur. Another drawback of MSCs is their inability to specify unwanted scenarios (anti-scenarios). We want to forbid the occurrence of these anti-scenarios, and they are crucial in setting up safety requirements.

Due to the weakness of the previous models, a new language for system specification, called Live Sequence Chart (LSC) was proposed in 1998 by Damm and Harel [17]. Later, in another work of Harel together with Marelly [24], the *Play-Engine* tool was presented allowing the modeling of LSC scenarios and also permitting the simulation of these scenarios through an executable model that does not need the source code. As the name suggests, LSCs specify liveness, things that must occur. They let modelers distinguish between possible and necessary behavior and also make possible to specify anti-scenarios.

LSC language fills out the gaps of the previous models, distinguishing things that can happen from those that must happen, through the use of some types of diagrams. Sequence of events that can happen in an execution of the system can be specified using existential chart that serves as a system test case. On the other hand, sequence of events that should happen for all and any execution of the system should be modeled using universal charts. Each universal chart possesses a pre-condition (prechart) that, if successfully executed, forces the execution of the scenario specified in the chart body, that if not satisfied, indicates a requirement violation.

Besides the simulation, the requirements validation process can be made by formal specification methods, which allow the development of systems without ambiguities, through well defined syntaxes and semantics. With such formal models, it could:

- accomplish mathematical verification that guarantee that models possess the requested properties;
- analyze if the proposed solution is acceptable under the performance point of view, indicating best strategies for implementation;
- accomplish the software development and improve the reliability about correct implementations (generation of correct code).

Several class of formal specification models have been proposed, among them algebraic methods, process algebras, logic based methods and Petri Nets.

CSP [25], CCS [38] and LOTOS [9, 54] are examples of process algebras used to model concurrent processes. They are languages for the specification of processes and the formulation of statements about them, together with calculations for the verification of these statements.

Petri Net (PN) [39, 41] is a family of formal models suit for representing synchronization, concurrency or resource sharing. Actually, Petri Nets was the first model for formally describing concurrent systems [46]. The graphic representation of a PN structure consists of elements connected by directed arcs. There are two types of elements vertexes, places represented by circles and transitions (rectangles). An arc connects places to transitions or transitions to places. A PN is a multi-graph, since it allows multiple arcs from an element to another.

Coloured Petri Nets (CPNs) [28, 29] are high-level PNs that support complex data types and hierarchy. CPNs combine the strengths of ordinary PNs with the strengths of a high-level programming language. PNs provide the primitives for process interaction, while the programming language provides the primitives for the definition of data types and the manipulations of data values.

## 1.2 Our Approach

Early system design modeling allows error detection due to imperfection in the design process as well as those related to requirement analysis phase. Therefore, preventing larger and costly problems due to late detection, especially with those embedded systems that must be correct, robust and efficient (critical embedded systems).

Modeling processes of real time systems must take into account both functional and nonfunctional requirements. Some models, as MSC and UML Sequence Diagram, seemed to supply such needs, however they possess some deficiencies:

- they are unable to verify critical properties of a system;
- they do not allow performance evaluation to be accomplished;
- they do not allow specify scenarios that should happen for all system run (liveness);
- they do not allow the modeling of anti-scenarios.

Due to the costs and the complexity of embedded systems, it is essential a system specification in a high-level of abstraction, where functional and architectural descriptions validation are necessary. Validation can be carried out by simulation or complemented through formal analysis/verification that is quite attractive because they spare exhausting simulations.

Live Sequence Chart (LSC), reduces some shortcomings inherent to MSC based models, such as allowing the possibility of specifying liveness and anti-scenarios. LSC allows modelers distinguish between possible and necessary behavior and specify anti-scenarios.

As well as the previous models (MSC and UML), LSC language possesses some deficiencies. It is not possible to verify system properties and to accomplish any system performance evaluation. So, if one wants to detect and reduce some risks that may lead to project failure, a formal approach, like PNs, could be used to allow execute such tasks.

As LSC language uses object oriented notions, in order to provide a faithful representation of LSC charts, this work uses a PN variant, called Coloured Petri Nets (CPNs), due to the possibility of representing complex data types. Besides this advantage, CPN models can be evaluated in many different ways.

The first evaluation method is interactive simulation. It is very similar to debugging and prototyping. This means that we can execute a CPN model, to make a detailed investigation of the behavior of the modeled system.

The second method is automatic simulation which is similar to program execution. It allows a fast execution of thousands or millions of transitions. The purpose is to investigate the functional correctness of the system or to investigate the performance of the system, e.g. to identify bottlenecks, to predict the use of buffer space or the mean/maximal service time.

The third method is based on the analysis of reachability graphs. The reachability graph is a directed graph which has a node for each reachable system state and an arc for each possible state change. Obviously, such a graph may become very large, even for small CPNs. However, it can be constructed and analyzed totally automatically, and there exist techniques which makes it possible to work with condensed occurrence graphs without losing analytic power.

The fourth analysis method is place invariants. This method is very similar to the use of invariants in ordinary program verification. The user constructs a set of equations which is verified for all reachable system states. The equations are used to verify properties of the modeled system, e.g., absence of deadlock.

## 1.3 Goals

The goal of this work is to aid the development process of embedded systems, through an approach that allows to verify the correctness and robustness of such systems, through an analysis and verification of properties of the modeled system.

Due to embedded system characteristics, its functional and architectural descriptions should be validated, reducing risks that may lead the design of an embedded system to failure. So, in order to provide a validation mechanism, this work presents a methodology for mapping the Live Sequence Chart (LSC) language to an equivalent Coloured Petri Net (CPN) model as an approach for analysis and verification of embedded systems' properties. As LSC language has data-types and adopts high-level concepts such as method invocation, Coloured Petri Nets have been adopted as a suit Petri net variant since it supports complex data-types, annotations, hierarchy as well as have an associated programming language (CPN-ML) that improves value's handling. Therefore, the proposition of a CPN model for LSC allows verification and analysis of systems described in LSC, hence, contributing for increasing designers' confidence on the system development process and reducing risks that may lead to project failure.

## 1.4 Related Works

Some works have been published as an approach to properties analysis and performance evaluation of system's specifications, starting from scenarios, which are described using a requirement specification language, such as UML, MSC, LSC and so on. Some related works are presented next.

The growing popularity of sequence charts, first of all Message Sequence Charts and UML Sequence Diagrams, for the description of communication behavior has evoked criticism regarding the semantics of the charts which led to extensions of these standardized visual formalisms. One such extension are Live Sequence Charts which allow to distinguish mandatory and possible behavior in protocol specifications. In the original language definition for LSCs the semantics are only described informally, although a sketch for a possible formalization has been provided as well. Klose and Wittke [31] intend to fill in the semantic blanks of the original LSC definition. Following the sketched path they define the semantics of an LSC by deriving a Timed Buchi Automata from it. They also consider qualitative and quantitative timing aspects. They finally show how LSCs are integrated into a verification tool set for *Statemate* designs.

Merseguer [26] proposes an approach to analyze performance for mobile agents systems, in which security and performance are the most critical aspects. This approach maps Message Sequence Charts (MSC) to a Generalized Stochastic Petri Nets (GSPN) model, through which, performance indexes may be computed by applying quantitative analysis techniques already developed in the literature. This approach proposes a UML with performance annotation (pa-UML) to model performance on these kind of systems. The problem domain is modeled using pa-UML, describing static and dynamic views when necessary. Through pa-UML, it is obtained the corresponding formal model expressed as Petri Nets.

Baresi [8] proposes High-Level Timed Petri Nets (HLTPN) for UML dynamic models in order to obtain a flexible and customizable representation to dynamic aspects of object-oriented models, in order to simulate particular parts of these models and if necessary analyze them. The proposal describes the main UML elements with formal semantics in terms of functionally equivalent to HLTPNs and shows results from execution and analysis as decorations to UML symbols.

Live Sequence Charts (LSCs) are a promising graphical specification formalism, usually applied to software systems. Bunker and Slind [12] adapt LSCs for the purpose of hardware requirements specification and verification. The main contribution of their work is an algorithm for generating temporal logic formulas from an LSC. The generated formulas are used as specifications for a model checker to verify compliance of hardware implementations.

The problem of relating state-based intra-agent (or intra-object) behavioral descriptions with scenario-based inter-agent (inter-object) descriptions has recently focused much interest among the software engineering community. Bomtemps and Heymans [55] investigate this problem. As inter-agent formalism, they adopt a simple variant of Live Sequence Charts. For the intra-agent perspective, they consider a game-theoretic foundation, looking at agents as "strategies" which encompasses the popular "state-based" paradigm. Three classes of relationships between models are studied: scenario checking (called eLSC checking), synthesis, and verification. They set a formally defined theoretical stage that allows to express these three problems very simply, to discuss their complexity, and to describe optimal solutions. Their study reveals the intrinsic high computational difficulty of these tasks. Consequently, many related problems and solutions are surveyed, some of which can be the basis for practical solutions. In this, we also offer a panorama of current research and directions for the future.

Bontemps [10] proposes an approach to obtain an automata representation from High-Level Live Sequence Charts (HLSC). This work builds automata from HLSC scenarios and show that standard algorithms on this (low-level) formalism can be used to check consistency and refinement, and to synthesize a state-based specification from a set of consistent requirements. The disadvantage of this approach is the description of the system that is given as an automata, which is difficult to read, and thus, of little interest for the later stages of development.

Kluge [32] focuses on Petri Nets as a formal model for analysis and simulation of Message Sequence Charts (MSC). Additionally, it proposes to use this Petri Net based formal model as a formal semantics for MSC. This approach provides a formally precise as well as an intuitive semantics for MSC. A further advantage of this approach is, that existing algorithms, methods and tools for analysing and simulating Petri nets can be employed for the analysis and simulation of MSCs. A drawback of this approach is that it is necessary changes of an MSC specification in order to derive a low-level Petri Net with the correct behavior.

Kugler and Harel [20] provide semantics for the powerful scenario-based language of live sequence charts (LSCs). They show how the semantics of live sequence charts can be captured using temporal logic. This is done by studying various subsets of the LSC language and providing an explicit translation into temporal logic. They show how a kernel subset of the LSC language (which omits variables, for example) can be embedded within the temporal logic CTL. For this kernel subset the embedding is a strict inclusion. They show that existential charts can be expressed using the branching temporal logic CTL, while universal charts are in the intersection of linear temporal logic and branching temporal logic.

Sun and Dong [52] investigate theoretical relations between LSCs and CSP. LSCs are formalized using trace and failure semantics so as to facilitate the semantic transformation from LSCs to CSP. The practical implication is that mature tool supports for CSP can be reused to validate LSCs.

Verification and validation are critical and costly for high-assurance systems. Even though many formal specification techniques are available to verify various properties for embedded systems, it takes much effort to develop the state model and specify properties using temporal logic. Tsai and Yu [53] present a process to rapidly generate the state model by simulating system scenarios, and formal model checking techniques can then be applied to the state model to verify various properties. Because system scenarios are widely used during embedded system development, the effort needed to develop the state model for the embedded system is thus greatly reduced. Their work present how informal system scenarios can be formalized and used in simulation to generate the state model. The simulation tool developed is also capable of performing runtime checking such as completeness and consistency checking, and timing analysis. The state model generated can be mapped to UML's state chart. Furthermore, they use a pattern based approach to specify properties to be checked rapidly. In this way, various formal model checking techniques can be applied to the embedded system development.

## 1.5 Dissertation Structure

This dissertation presents a methodology for representing LSC diagrams by a Coloured Petri Net (CPN). The CPN model is considered for analysis and verification of qualitative properties. Thus, taking an important concern in embedded system design process.

Chapter 2 addresses some specification models and a formal approach that can be used to analyze and verify the behavior of a specified system. In Chapter 3, the mapping of the LSC language to an equivalent CPN model are presented. Chapter 4 presents two case studies, on which the mapping is applied in order to analyse and verify some properties. Finally, a conclusion and future works are presented in Chapter 5.

# Chapter 2

# Basic Concepts

*This chapter presents some details on object-oriented analysis and design, mentioning some specification models with their advantages and disadvantages in the system modeling process. Finally, in order to allow properties analysis and verification, PNs are presented as a possible approach to formal verification.*

## 2.1 Object-Oriented Analysis and Design

The late 1980s saw the first proposals for object-oriented analysis and design (OOAD). As in structured analysis/structured design (SA/SD), the basic idea in modeling system structure was to lift concepts up from the programming to the modeling level and to use visual formalisms. Inspired by entity-relationship diagrams [15], several methodology teams recommended various forms of class and object diagrams for modeling system [11, 16, 51]. To model behavior, most object-oriented modeling approaches adopted state charts [22]. Each class has an associated state chart, which describes the behavior of any object instance.

The issue of connecting structure and behavior is subtler and more complicated in the OOAD world than in the SA/SD world. Classes represent dynamically changing collections of concrete objects. Behavioral modeling must thus address issues related to object creation and destruction, message delegation, relationship modification and maintenance, aggregation, inheritance, and so on.

The links between behavior and structure must be defined in sufficient detail and with enough rigor to support the construction of tools that enable model execution and full code generation. Only a few tools have been able to do this. One is Object-Time, which is based on the Real-Time Object-

Oriented Modeling method [7] and is now part of the Rational RealTime tool [6].

Another tool is Rhapsody [5], which is based on the work of Gery and Harel of executable object modeling with state charts [23]. This work centers on a carefully constructed language set that includes class/object diagrams adapted from the Booch method [11] and the OMT method [51], driven by state charts for behavior.

This pair of languages also serves as the executable heart of the Unified Modeling Language (UML) [18], put together by a team led by Grady Booch, James Rumbaugh, and Ivar Jacobson, which the Object Management Group (OMG) adopted as a standard in 1997 [40]. The class/object diagrams and the state charts part of the UML is often called XUML. Thus, XUML is the part of UML that specifies unambiguous, executable, and therefore implementable, models.

## 2.2 Message Sequence Chart (MSC)

The language of message sequence charts (MSCs) is a popular mechanism for specifying scenarios that describe possible interactions between processes or objects. MSCs are particularly useful in the early stages of system development. The language has found into many design methodologies, and a variant of it has been made part of UML, where it is called sequence diagrams. There is also a standard syntax for MSCs that appears as a recommendation of the ITU [14].

In many object-oriented system development methodologies, the user first specifies the system's use cases and some specific instantiations of each use case are then described using sequence diagrams (MSCs). In a later modeling step, the behavior of a class is described by a state diagram [22] that prescribes a behavior for each of the instances of the class. Finally, the objects are implemented as code in a specific programming language. Parts of this design flow can be automated, such as the generation of code from object model diagrams and state charts, as exemplified in ObjecTime [42] and Rhapsody [23, 5].

In such design flows, the main role of MSCs is to capture system requirements in the form of "good" scenarios that the implemented system should exhibit. Sometimes an MSC is prepared for a "bad" scenario that the implementation should not allow. System requirements captured in this intuitive fashion can serve as a useful interface between the end-users of the system and the system designer. They can also serve as a test for validating some aspects of the implementation. A substantial portion of research on

MSCs has been driven by this way of using MSCs, with the focus on mechanisms for describing collections of scenarios, techniques for analyzing such collections and relating them to a state-based executable specification.

Figure 2.1 depicts a simple MSC chart. This chart captures a scenario in which a user (U) sends a request to an interface (I) to gain access to a resource (R). The interface in turn sends a request to the resource, and receives “grant” as a response, after which it sends “yes” to the user. The vertical lines represent the life-lines of the processes taking part in the scenario . As usual, time is assumed to flow downwards along each life-line. The directed arrows going across the life-lines represent the causal link from a send event (the source of the arrow) to the corresponding receive event (the target of the arrow), with the label on the arrow denoting the message being transmitted.

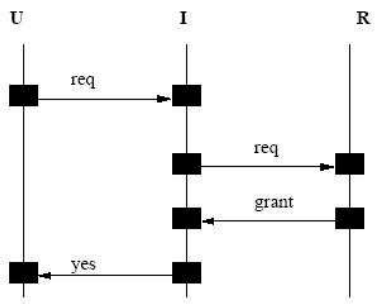


Figure 2.1: Simple MSC chart

An MSC chart is guided by the following rules:

- all the events that a process takes part in are linearly ordered, each process is a sequential agent;
- messages must be sent before they can be received;
- there are no dangling communication edges in an MSC, therefore all sent messages have also been received;
- the causality relation between the events in an MSC is completely determined by the order in which the events occur within each process and communication relation relating a send-receive pairs.

## 2.3 Live Sequence Chart (LSC)

LSC [17] is a system specification language based on scenarios that allows to specify anti-scenarios as well as it permits specify what should happen for all system runs. LSC language fills out the gaps of the previous models, distinguishing things that can happen of things that must happen, through the use of some types of diagrams. Sequence of events that can happen in an execution of the system can be specified using existential chart that serves as a system test case. On the other hand, sequence of events that should happen for all and any execution of the system should be modeled using universal charts. Each universal chart possesses a pre-condition (prechart) that, if successfully executed, forces the execution of the scenario specified in the chart body. Otherwise it indicates a requirement violation.

An LSC specification is formed by many scenarios, that can be specified using universal charts or existential charts.

**Definition 2.3.1.** *(Specification) An LSC specification $S$ for a system $Sys$ is defined as a disjoint union $S = S_U \cup S_E$, where $S_U$ is a set of universal charts and $S_E$ is a set of existential charts.*

Figure 2.2 depicts an LSC universal chart that contains three instances, *User*, *MainSwitch* and *MainLight*. This diagram specifies that every time the *User* modifies *MainSwitch* to *On*, then the instance *MainLight* should sets itself to *On*. *User* and *MainSwitch* participate in prechart (denoted by dashed lines) and chart body (denoted by solid lines) scenarios. *MainLight* participates in chart body scenario only. An instance is participating in a scenario, when its instance line (vertical line) is present inside the scenario's scope.

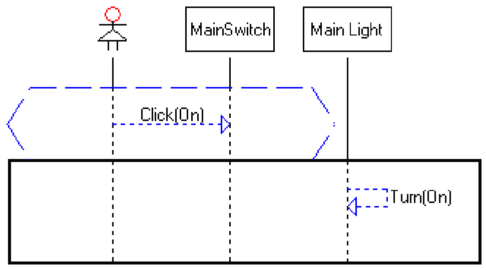


Figure 2.2: Simple LSC chart

LSC language possesses a vast number of constructions that can be used inside a chart, such as messages, conditions, assignments, loops, *if-then-else* construction, forbidden elements, time restrictions.

**Definition 2.3.2.** *An LSC chart L is defined to be: L = ($I_L$, $V_L$, $M_L$, $[Pch_L]$, $A_L$, $C_L$, $SUB_L$, $ITE_L$, $LOOP_L$, $\bar{M}_L$, $\bar{C}_L$, Strict, event, subchart, temp), where $I_L$ is the set of LSC instances, $V_L$ is the set of variables used in L, $M_L$ is the set of messages in L. A message $M_i \in M_L$ is represented by a directed "arrow" (graph), with a location point on an instance line that represents the sending event and other location point on an instance line (can be the same instance line) that represents the receiving event. $Pch_L$ is the prechart of L (in universal charts), which is optional, $A_L$ is the set of assignments in L, $C_L$ is the set of conditions in L, $SUB_L$ is the set of subcharts in L, $ITE_L$ is the set of if-then-else constructions in L, $LOOP_L$ is the set of loops in L, $\bar{M}_L$ is the set of forbidden messages in L, $\bar{C}_L$ is the set of forbidden conditions in L, , Strict is a boolean flag indicating whether the LSC is strict or tolerant, temp function assigns temperature to locations, messages, conditions, forbidden messages and forbidden conditions, event function maps a location to the event it is associated with, and subchart is a function that returns the corresponding subchart for a particular location.*

Despite the visual, LSCs constitute a formal language, which will not always be appropriate for the people involved in the early stages of requirement capturing. So, a higher-level approach to the problem of specifying scenario-based behavior, termed *play-in* scenarios, was proposed and briefly sketched, together with the *Play-Engine* tool [24] that supports it.

The main idea of the *play-in* process is to raise the level of abstraction in requirements engineering, and to work with a look-alike version of the system under development. This enables people who are unfamiliar with LSCs, or who do not want to work with such formal languages directly, to specify the behavioral requirements of systems using a high-level, intuitive and user friendly mechanism. These could include domain experts, application engineers, requirements engineers, and even potential users.

What *play-in* means is that the system's developer first builds the GUI (interface) of the system, with no behavior built into it. In systems for which there is a meaning to the layout of hidden objects, the user may build the graphical representation of these objects as well. In fact, for GUI-less systems, or for sets of internal objects, we simply use the object model diagram as a GUI. In any case, the user "plays" the GUI by clicking buttons, sending messages (calling functions) to hidden objects in an intuitive drag & drop manner. With an object model diagram as the interface, the user clicks the objects and/or the methods and the parameters. By similarly playing the GUI, the user describes the desired reactions of the system and the conditions that may or must hold. As this is being done, the *Play-Engine* continuously constructs LSCs automatically. It queries the application GUI (that was built by the user) for its structure, and interacts with it, thus manipulating the information entered by the user and building and exhibiting the appropriate LSCs.

After *playing in* the specification, the natural thing to do is to verify that it reflects what the user intended to say, and here is where the *play-out* mechanism enters, allowing to test and validate the requirements as well. In *play-out*, the user simply plays the GUI application as he/she would have done when executing a system model, or the final system, but limiting him/herself to "end-user" and external environment actions only. While doing this, the *Play-Engine* keeps track of the actions and causes other actions and events to occur as dictated by the universal charts in the specification. Here, the engine interacts with the GUI application and uses it to reflect the system state at any given moment, with no intra-object model having to be built or synthesized. This makes it very easy to let all kinds of people participate in the process of debugging the specification, since they do not need to know anything about the specification or the language used. It yields a specification that is well tested and which has a lower probability of errors in later phases, which are a lot more expensive to detect and eliminate.

In the LSC language, the system is modeled using object oriented notions and terminologies. So, a system is composed of objects that represent instances of a given class, which are formed by properties based on some application data type. These objects can be created in an independent way or they can be based on another existent object, inheriting their characteristics and methods.

A property has a name. It is identified by a unique ID and it is based on a data type with a certain domain, starting from the value for the property is chosen.

In order to provide an intuitive and user-friendly support, *Play-Engine* tool [24] requires object properties to also have the following characteristics:

- *Prefix* - It is a verb used to describe the action of changing the property's value;

- *IsDefault* - If a property is default, its name is not shown in the LSC message;

- *InOnly* - Object properties are usually changed by either operating the object, or by right-clicking the object and choosing a value to an object's property. An *InOnly* property can be changed only by using the first of these;

- *Can be changed externally* (*ExtChg*) - This indicates that the property can be changed by the system's environment;
- *Affects* - When the value of an object's property is changed, a message is drawn (a directed arrow as seen in Definition 2.3.2) in the LSC chart. The value of the *Affects* flag shows how this arrow is drawn and the possibilities are *User*, *Env* and *Self*, and the arrow is drawn toward the user, toward the environment, or as a self arrow, respectively;
- *Synchronous* (*Sync*) - A synchronous message may be propagated only if both the sender and receiver are ready.

**Definition 2.3.3.** *(Property) An object property P is defined as P = (Name, D, InOnly, ExtChg, Affects, Sync, Prefix, IsDefault), where Name is the property name, D is a finite set of possible property's values and Prefix is a verb used to describe the action of changing the property's value. InOnly, ExtChg, Sync and IsDefault range over {true, false} and Affects ranges over {User, Env, Self}.*

**Definition 2.3.4.** *(Class) A class C is defined as C = (Name, CP, SM), where Name is the class name, CP is the set of class properties and SM is the set of class methods.*

Each object may be an instance of some defined class. An object that is not explicitly associated with a class is considered to be the single object.

**Definition 2.3.5.** *(Object) Let O be the set of concrete objects. An object $o_i \in O$ is a concrete object of some class and is therefore defined as $o_i$ = (Name, C, PV), where Name is the object's name, C is its class name and PV is a function that assigns a value to each of object's property.*

Application data types can be created starting from primitive types, such as:

**Enumerated** defines a finite group of values. For example, *Week* = *Sun*, *Mon*, *Tue*, *Wed*, *Thu*, *Fri*, *Sat*;

**Discrete** defines a minimum value, a maximum value and a step that determines the interval between consecutive values. For example, *Byte* can be represented by a discrete type ranging from 0 to 127 with a step of 1;

**String** defined by a maximum length.

A LSC system specification defines the set of application types, the set of classes and the set of externally implemented functions that can be used in any scenario of the specification. Coupled to this system specification a global clock is used to check time restrictions that can be imposed inside a LSC scenario.

**Definition 2.3.6.** *(LSC model) An LSC model is defined as $Sys = (AT, C, O, F, Clock)$, where $AT$ is the set of application data types, $C$ is the set of classes, $O$ is the set of objects, $F$ is the set of externally implemented functions and $Clock$ is the system global clock.*

Inside of an LSC specification two types of charts can be used: universal and existential. The first is denoted by a solid border line and it is used to specify restrictions that are applied for all system runs. The last is denoted by a dashed border line and it can be used to specify system tests, which are applied to at least one system's execution.

Each universal chart has a pre-condition, called *prechart*. If this pre-condition is successfully executed then the chart body should be satisfied by the system. In that way, an universal chart establishes an action-reaction relationship between the *prechart* and the chart body.

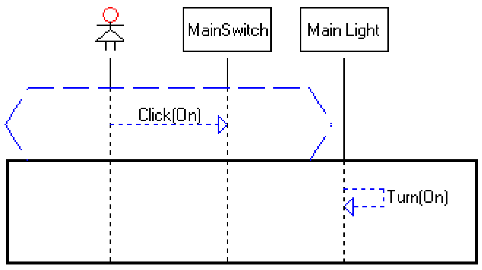


Figure 2.3: An universal chart

Figure 2.3 shows an universal chart. This chart says that whenever the instance *User* sets *State* property (*IsDefault* is true) of *MainSwitch* to *On*, then *MainLight* must set its *State* property (*IsDefault* property is configured as true) to *On*.

Figure 2.4 shows an existential chart. It is necessary to observe that there is no execution order between events, therefore, it is possible a scenario in which *MainLight* is turned on (*Turn(On)*), and then the *User* sets *MainSwitch* to *On* (*Click(On)*).

A system specification has many LSC charts, which represent LSC scenarios where some system's functionalities are modeled using a large num-

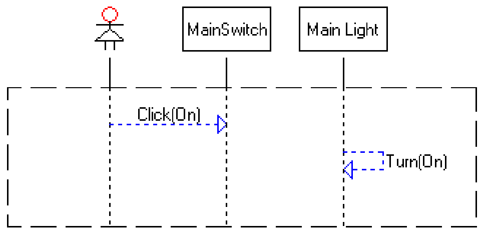


Figure 2.4: An existential chart

ber of available LSC constructions, such as messages, conditions, assignments, loops, *if-then-else* constructions, forbidden elements, time restrictions. Some of these constructions will be shown later.

An LSC chart has several instances attached to it, which are the representation of a concrete object (Definition 2.3.5). Every instance line (vertical line starting from the rectangle that represents the instance) contains locations. An instance progresses from one location to the next by participating in some activity associated with the location. Such activity could, for example, be the sending or receiving of a message. Every instance has also an initial location and a final location, in which the instance begins and terminates, respectively.

Each location has a "temperature" that can be *hot* or *cold*. A *hot* location forces the instance to progress throughout its instance line, while a *cold* location allows the instance to stay in this location without violating the chart.

An LSC event is an action that occurred inside of a chart, which consists of two disjoint sets. An LSC event can be an actual system event of sending or receiving a message, or it can be one of the acts of entering the prechart, exiting it, entering the chart body or reaching its end. The first kind of event is called a visible event and the second is called a hidden event.

**Definition 2.3.7.** *(Events) Let $E_L$ be the set of LSC events. An LSC event $e_i \in E_L$ is an action that occurred inside of a chart L, such as messages, conditions, assignments and synchronization points, described as a tuple $[Pos, Location]$, where $Pos$ may represent the scenario (prechart, chart body or subchart) in which the event occurred or it may represent a message inside the chart L, and $Location$ represents the event's location, which can be $Start$, $End$, $Send$ and $Recv$. $Start$ and $End$ are used when $Pos$ represents a scenario, and they denote beginning and ending of the scenario, respectively. $Send$ and $Recv$ are used when $Pos$ represents a message, and they denote sending and receiving event of the message, respectively.*

**Definition 2.3.8.** *(Functions loc and evnt) Let $l_L$ be the set of locations of a chart L. An LSC event may have cold or hot locations, therefore it can define a function $evnt : l_L \mapsto E_L$ that maps each location into the event it is associated with, as well as its inverse, $loc : E_L \mapsto 2^{l_L} = evnt^{-1}$ that maps an event into the set of locations associated with it. $2^{l_L}$ is applied because a location can be cold or hot.*

A LSC event may be visible or hidden, with locations associated with it. A hidden event may be entering or exiting a prechart, a chart body or a subchart. So by applying the function $loc$ for the events $[Pch, Start]$, $[Pch, End]$, $[CB, Start]$, $[CB, End]$, $[Sub, Start]$, $[Sub, End]$, it returns, respectively, the locations associated with the beginning of the prechart, the locations associated with the ending of the prechart, the locations associated with the beginning of the chart body, the locations associated with the ending of the chart body, the locations associated with the beginning of a subchart, and the locations associated with the ending of a subchart.

An object instance is a representation of a concrete object of a certain class. Each instance has a set of locations, which indicate events in which the instance is participating in the scenario. Besides this, an instance has a set of bind expressions that compare their properties' values, a reference for the chart's type and a set of prohibited elements which the instance is not allowed to bind to.

**Definition 2.3.9.** *(Instance) An instance I is defined as $I = (l, O, \psi, Mode, \phi)$ where $l$ is the set of instance locations, $O$ is the concrete object represented by I, $\psi = \{\psi_i\}$, $\psi_i = \{(p,\ o,\ r) \mid p \in P,\ o \in Oper,\ r \in RHS\}$ is a binding expression of the set of binding expressions $\psi$, where $P$ is a set of instance properties, $Oper$ is a set of relational operators and $RHS$ can be a constant value, a variable or a function call, $Mode \in \{Existential, Universal\}$, and $\phi$ is a set of forbidden objects, which the instance is not allowed to bind to. We denote by $l_x^i$ the $x^{th}$ location of instance $i_i$, and by $i_i.l$ the set of locations of instance $i_i$.*

Inside a LSC chart, it is possible to delimit scenarios using *subcharts*. A subchart is a chart's fragment, denoted by a rectangle with a solid border line, which encloses all the participant instances.

Figure 2.5 shows a subchart with two constructions, the condition *TermSelect.Therm=1* and the assignment *Tc:=Thermo1.Temp*. These constructions will be presented later in this work. Only instances *TermSelect* and *Thermo1* are participating in the scenario of the presented subchart.

Alike prechart and chart body, the beginning and ending of a subchart are synchronization points. Hence, every participant instance of a subchart

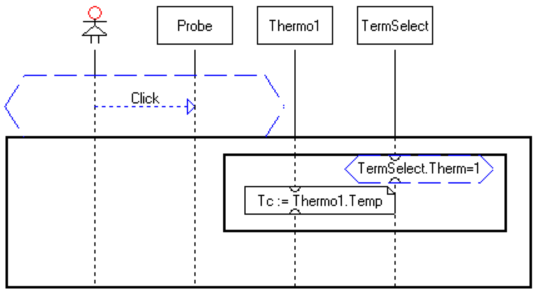


Figure 2.5: An LSC subchart

enter and exit its scenario simultaneously. Besides, it can be attached any of the construction that can be inserted in the prechart or chart body, such as messages, conditions, assignments, loops, if-then-else.

Let $M_L$ be the set of messages in chart $L$, $A_L$ be the set of assignments in chart $L$, $C_L$ be the set of conditions in chart $L$, $I_L$ be the set of instances in chart $L$, $I_A \subseteq I_L$ be the set of instances involved in some activity with an assignment $a_i \in A_L$, $I_C \subseteq I_L$ be the set of instances involved in some activity with a condition $c_i \in C_L$, $SUB_L$ be the set of subcharts, $subchart : M_L \cup A_L \cup C_L \rightarrow SUB_L$ be a function that returns for each construction the subchart to which it belongs, and $Sub_i \in SUB_L$ be a subchart in chart $L$. The set of instances, $I_{Sub_i}$, participating in a subchart $Sub_i$ is defined as the set of all instances that are involved in some activity in $Sub_i$: $I_{Sub_i} = \{i_i \in I_L | \ \exists m_i \in M_L$ s.t. $(subchart(m_i) = Sub_i \wedge (i_i = m_i.i_{Src} \vee i_i = m_i.i_{Dst})) \vee \exists \ a_i \in A_L$ s.t. $(subchart(a_i) = Sub_i \wedge i_i \in I_A) \vee \exists \ c_i \in C_L$ s.t. $(subchart(c_i) = Sub_i \wedge i_i \in I_C)\}$, where $m_i.i_{Src}$ is the object sending the message and $m_i.i_{Dst}$ is the object receiving the message.

Alike locations, messages in the LSC can be *hot* or *cold*. A *hot* message must be received after sent and the *cold* one can be sent and not received. *Hot* messages are denoted by red solid lines, while *cold* messages are denoted by blue dashed lines. Through this work, we consider that a *cold* message eventually arrive after it is sent. Figure 2.6 depicts some *hot* and *cold* messages.

Besides the "temperature" that can be applied to messages, messages can represent a synchronous or asynchronous communication. A synchronous message is denoted by an arrow with a closed triangle and the asynchronous message is represented by an arrow with an open triangle. The messages *Thermo1.Change(30)* and *Console.Show(Cold Oven!)*, presented in Figure 2.6, are asynchronous and synchronous, respectively.

A message can have as sender or receiver, the *User*, the *Environment*,

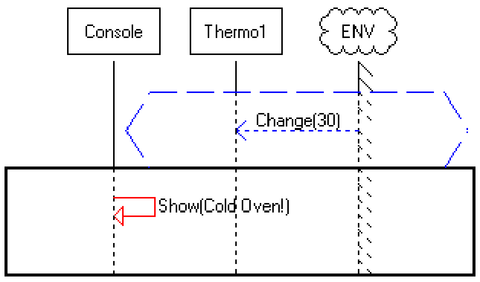


Figure 2.6: *Hot* and *cold* messages

other objects or the *Clock* object that represents the global clock, which has the *Time* property that returns the current time and the *Tick* method that increments a time unit in the current time. When the sender and receiver are the same, it is called a *self* message .

Every message has two locations, one for the sender and other for the receiver, therefore the steps to obtain the CPN model should be applied for both. Let $m_i \in M_L$ be an LSC message, so by applying the function $loc$ for the events $[m_i, Send]$ and $[m_i, Rcv]$, it returns, respectively, the location at the sending point of message $m_i$ and the location at the receiving point of message $m_i$.

A system message can set a property value of an object or just pass the value ahead, as a method call. Each system message has a sender and receiver. If the message represents a modification of property value, then the property that has its value modified belongs to the receiver instance. Those messages can inform the values through constants (exact) or through variables (symbolic) that allows the construction of more general scenarios.

The *Play-Engine* tool [24] allows that applications with graphical interface supply external functions, which are identified by a name, name and type of the parameters and the type of the result. *Play-Engine* tool can interact with a GUI application and request a function by passing the expected parameters and receiving the returned value.

**Definition 2.3.10.** *(External function) Let $AT$ be the set of LSC application types (domains), $D \in AT$ be a finite set of values. An external function is defined as $Name : d_1 \times d_2 \times ... \times d_n \rightarrow d_F$, where $Name$ is the function name, $d_i \in D$ is the type of its $i^{th}$ formal parameter, and $d_F \in D$ is the type of its returned value.*

**Definition 2.3.11.** *(Function information structure) Let $F$ be the set of external functions. A function information structure $\lambda_f^F$ is defined for $f \in F \cup \{\perp\}$ as follows:*

- $\lambda_f^F = \begin{cases} (v_1 \in f.d_1, ..., v_n \in f.d_n), & \textit{if } f \in F; \\ \bot, & \textit{if } f = \bot. \end{cases}$,

  *where $v_1, ..., v_n$ are variables which represent formal parameters of a external function and $\bot$ represents an absence of information.*

**Definition 2.3.12.** *(Object method) Let $AT$ be the set of LSC application types (domains), $D \in AT$ be a finite set of values. An object method $M$ is defined as $(Name(d_1, d_2, ..., d_n), Sync)$, where $Name$ is the method name, $D_i \in AT$ is the type of $i^{th}$ formal parameter, and $Sync \in \{True, False\}$ indicates whether calling this method is a synchronous operation.*

**Definition 2.3.13.** *(Method information structure) Let $C$ be a class and $C.SM$ be the set of class methods. A method information structure $\lambda_m^M$ is defined for $m \in C.SM \cup \{\bot\}$ as follows:*

- $\lambda_m^M = \begin{cases} v_1 \in m.d_1, ..., v_n \in m.d_n, & \textit{if } m \in C.SM; \\ \bot, & \textit{if } m = \bot. \end{cases}$,

  *where $v_1, ..., v_n$ are variables which represent formal parameters of a method call and $\bot$ represents an absence of information.*

**Definition 2.3.14.** *(System message) Given a system model $Sys$ (Definition 2.3.6), a system message $m_S$ is defined as $m_S = (P,\ V,\ f,\ \lambda_f^F,\ m,\ \lambda_m^M,\ Symbolic)$, where $P$ is the property changed, $V$ is a variable holding a new value for the property $P$, $f$ is a function describing a new value for $P$, $\lambda_f^F$ is a function information structure, $m$ is the method of $Dst$ called by $Src$, $\lambda_m^M$ is a method information structure, and $Symbolic$ is a boolean flag indicating whether the message is symbolic, where $(P \neq \bot) \vee (m \neq \bot)$ represents either a property change or a method call.*

A LSC message is a system message with two instances, one representing the sender and other representing the receiver.

The system message represents how the message is formed and presented inside the scenario, and the LSC message represents the event, indicating the sender and receiver of the message.

**Definition 2.3.15.** *(Message) Let $I_L$ be the set of instances of chart $L$ and $m_S$ be a system message. A message $m_i \in M_L$ is defined as $m_i = (i_{Src},\ i_{Dst},\ m_S)$, where $i_{Src} \in I_L$ is the instance representing the sender, $i_{Dst} \in I_L$ is the instance representing the receiver, and $m_S$ is the system message represented by $m_i$.*

In LSC scenarios the events are executed top-down. Each event has two locations, sending and receiving locations. At the synchronization points, instances' locations are executed at the same time. For the synchronous messages, sending and receiving locations are executed at the same, on the other hand at asynchronous messages the sending location has a precedence over the receiving location.

Let $i_i, i_j \in I_L$ be instances of chart $L$, $x, y \in \mathbb{N}$ be natural numbers, $l^i_x$ and $l^i_y$ be locations of an instance $i_i$, $l^j_y$ be a location of an instance $i_j$, $M^S_L \subseteq M_L$ be the set of synchronous messages, $M^A_L \subseteq M_L$ be the set of asynchronous messages. The function $first$ checks if it is the first occurrence of a variable in a chart. The function $affects$ checks if a variable is modified, and $uses$ is a function that checks if a variable is used but not modified. $l^i_x \leq L\ l^i_y$ denotes $l^i_x$ precedes $l^i_y$, and $l^i_x = L\ l^i_y$ denotes $l^i_x$ and $l^i_y$ are executed at the same time. The execution order is defined as follows:

- the locations along a single instance line are ordered top-down. Thus, things higher up are carried out earlier $x < y \Rightarrow l^i_x < L\ l^i_y$;
- for an asynchronous message $m \in M^A_L$, the location of $([m, Send])$ event precedes the location of the $([m, Rcv])$ event. For synchronous messages, two events take place simultaneously:
  $\forall m_i \in M^A_L : m_i \Rightarrow loc([m_i, Send]) < L\ loc([m_i, Rcv])$
  $\forall m_y \in M^S_L : m_y \Rightarrow loc([m_y, Send]) = L\ loc([m_y, Rcv])$
- all instances participating in the prechart and the chart body are synchronized at the beginning and at its end.
  $\forall l^i_x, l^j_y \in loc([Pch, Start]) : l^i_x = L\ l^j_y$
  $\forall l^i_x, l^j_y \in loc([Pch, End]) : l^i_x = L\ l^j_y$
  $\forall l^i_x, l^j_y \in loc([CB, Start]) : l^i_x = L\ l^j_y$
  $\forall l^i_x, l^j_y \in loc([CB, End]) : l^i_x = L\ l^j_y$
- all instances that participate in a subchart are synchronized at start and end.
  $\forall Sub_i \in SUB_L$
  $\forall l^i_x, l^j_y \in loc([Sub_i, Start]) : l^i_x = L\ l^j_y$
  $\forall l^i_x, l^j_y \in loc([Sub_i, End]) : l^i_x = L\ l^j_y$.
- the first location that affects a variable precedes all other locations that affect or use the variable.
  $\forall l, l' \in l_L : first(l, X) \wedge (affects(l', X) \vee uses(l', X)) \Rightarrow l < L\ l'$.
- all instances that participate in an assignment are synchronized there.
  $\forall a_i \in A_L, \forall l^i_x, l^j_y \in loc(a_i) : l^i_x = L\ l^j_y$.
- all instances that participate in a condition are synchronized there.
  $\forall c_i \in C_L, \forall l^i_x, l^j_y \in loc(c_i) : l^i_x = L\ l^j_y$.

When a diagram becomes active, the instances begin in their initial locations and progress in their instance lines while the execution continues until they reach their final locations. A *cut* contains the next location to be executed for each instance inside a chart. A *cut* is *hot* if at least one of the instances is in a *hot* location, and it is *cold* if every instance is in a *cold* location. Figure 2.7 presents a *cut* which is denoted by a hatched line.

For several *cuts* that exist during the execution of a diagram, there are events that can happen and events that if happen will cause a violation. An event that appears in a diagram is said enabled ($e_e$) if it appears immediately after the *cut*, in other words, all of the events that should have happened before, already have successfully happened. For example, in Figure 2.7, the event *Switch2.Change(Med)* appears immediately after the *cut* (hatched line), so it is enabled. If an event does not appear immediately after the *cut* it is violating event($e_v$). If a violating event happens in the prechart scenario then the diagram is interrupted without causing errors. It just indicates that the scenario was not successfully contemplated. The same happens with a violating event if it happens in the chart body while the *cut* is *cold*, because a *cold* location does not force an instance to progress. However, if this violating event happens in the chart body while the *cut* is *hot*, then the diagram is aborted indicating a violation. In Figure 2.7, the event *Switch1.Change(Med)* does not appear immediately after the *cut*, so it is a violating event at this point.

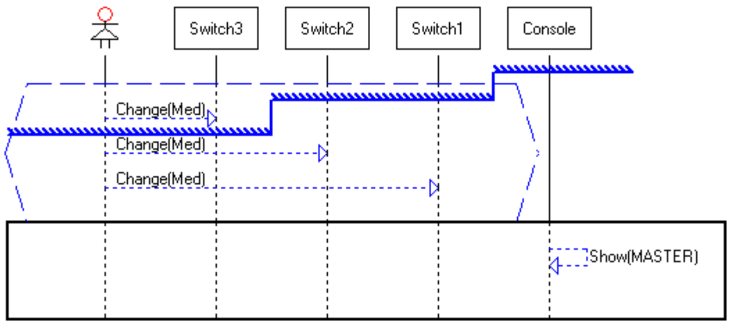


Figure 2.7: *User* sending a message

The *Play-Engine* tool [24] allows applications with graphical interface supply external functions, which are identified by a name, name and type of the parameters and the type of the result. *Play-Engine* tool can interact with a GUI application and request a function by passing the expected parameters and receiving the returned value.

The intention of this work is not to consider an equivalent model for the source code of these external functions. The code that will be executed must be considered as a “black box”, where it is enough to know that parameters will be passed and a value will be returned.

Figure 2.8 presents an example of an external function invocation. The message *Display.Show(X174 + X176)* makes a function call (*Show*) which uses the variables *X174* and *X176* as parameters.

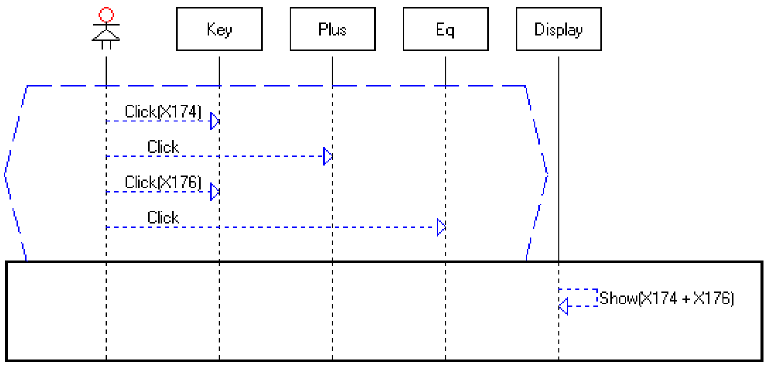


Figure 2.8: External function call

An assignment is an LSC construction that allows storing properties' values, constant values or a result of a external function, for a subsequent use inside the chart. The expression that is on the right side of the operator “:=” can be any of these mentioned values. On the left side of the operator, those values are stored in variables. Variables that are on the operator's left side are said affected by the assignment, while the ones that are on the right side are used variables.

An assignment may have several instances that synchronize their activities with it. None of these instances can continue beyond the assignment, until all of them have successfully executed their previous tasks. Synchronization points are represented by semi-circles that link the assignment to the participant instance line (see Figure 2.9).

Additionally, an assignment can be used to construct a time restriction that is a feature available in the LSC language, which permits to define time restrictions for real-time system's events.

**Definition 2.3.16.** *(Assignment) An assignment $a_i \in A_L$ is defined as $a_i = (V, I_A, C, P, f, \lambda_f^F, Timed)$, where $V$ is the variable, $I_A \subseteq I_L$ is the set of instances that are synchronized with the assignment, $C \in (\bigcup_{D \in AT} D) \cup \{\perp\}$ is a constant of some type in case the assignment stores a constant and $\perp$ if*

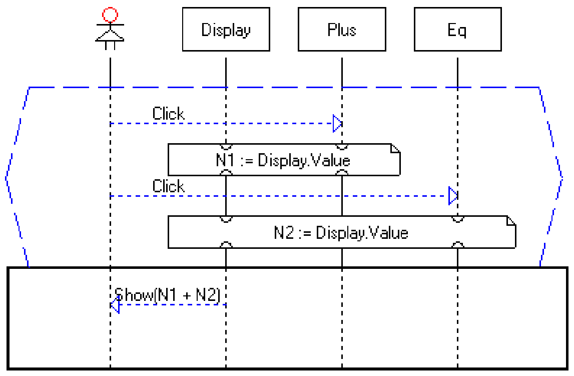


Figure 2.9: LSC scenario with assignments

*not,* $P \in (\bigcup_{I \in I_A} I.O.P) \cup \{\perp\}$ *is the property stored in case this assignment stores a property value and* $\perp$ *if not,* $f \in F \cup \{\perp\}$ *is a function in case the assignment stores some function and* $\perp$ *if not,* $\lambda_f^F$ *is a function information structure in case* $f \neq \perp$*, and* $Timed \in \{True, False\}$ *is a flag indicating whether* $a_i$ *is a timed assignment.*

A condition represents a decision structure that can be composed of a conjunction of expressions and can be evaluated as true or false.

A "temperature" can also be applied to conditions. A *cold* condition is denoted by a hexagon with a blue dashed lines. If a *cold* condition evaluates to *true*, then the execution of the diagram progresses to the next location after the condition, otherwise the diagram or the underlying subchart is abandoned. A *hot* condition is denoted by a hexagon with red solid lines. A *hot* condition should be evaluated to *true*, otherwise indicates a requirement violation. Figure 2.10 depicts *cold* and *hot* conditions.

Like assignments, a condition can be considered to synchronize several instances, i.e., a synchronized instance can not progress beyond the condition until all participating instances have reached the condition location. The instances that are synchronized with the condition have a semi-circle that links the condition to the participant instance line.

Figure 2.10 depicts two conditions, the *hot* condition $Light1 <> Green$, which is synchronized with the *Light1* instance and the *cold* condition *Light1*=*Green*, which is synchronized with instances *Light1* and *Console*.

A condition is composed of one or more of the following expressions:

- a basic expression that constrains a property or a variable, using some operator, with a constant value, another variable or with a function;

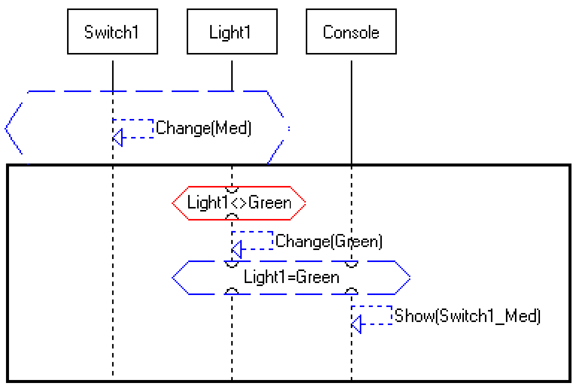


Figure 2.10: *Cold* and *hot* conditions

- a basic expression that consists of the reserved words *SYNC*, *TRUE*, *FALSE*, or the *SELECT* statement with probability;
- a basic expression that is a timing constraint, which constrains the current time with respect to a time value stored in some variable and a delay that can be a constant value, another variable or a function.

**Definition 2.3.17.** *(Condition) A condition $c_i \in C_L$ in an LSC chart $L$ is defined as $c_i = (I_C, \varphi, Timed)$, where $I_C \subseteq I_L$ is the set of instances that are synchronized with the condition, $\varphi$ is the set of basic expressions and $Timed \in \{True, False\}$ is a flag indicating whether $c_i$ is a timed condition.*

A condition uses all variables appearing in its expressions. Therefore, a condition can be executed whenever assigning values to all used variables. Besides the variables, the temperature of a condition may alter the moment in which this condition can be executed. In the case of a *cold* condition, it is immediately executed and may produce a true or false value. On the other hand, a *hot* condition is evaluated until having a true value. The execution will be stopped at this point if the specification is incorrect, because the value will never become true.

The *if-then-else* construction allows different scenarios to be executed depending on a condition. This construction consists of two adjacent subcharts, one that represents the *then* part and other that represents the *else* one, surrounding by a controlling condition at the top of the first subchart. The *else* part is not mandatory.

Figure 2.11 presents an example of an *if-then-else* construction. In this construction, the condition *Prb-Ctrl.Probing=True* defines which scenario should be executed. If this condition is true, then the scenario of *then* part

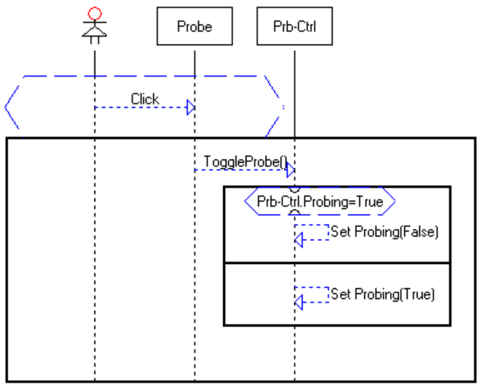


Figure 2.11: An *if-then-else* construction

is executed, so the message *Set Probing(False)* must occur. Otherwise, the message *Set Probing(True)* of *else* scenario is executed.

**Definition 2.3.18.** *(If-then-else construction) Let $SUB_L$ be the set of subcharts of a chart $L$. An if-then-else construction $ITE$ in a chart $L$ is defined as $ITE = (I_{ITE}, C, Sub_T, Sub_E)$, where $I_{ITE} = I_C \cup I_{Sub_T} \cup I_{Sub_E}$ is the set of instances participating in the if-then-else construction. $C$ is the main condition of the if-then-else construction, $Sub_T \in SUB_L$ is the subchart containing the then part, $Sub_E \in SUB_L \cup \{\perp\}$ is the subchart containing the else part (if there is no such part, $Sub_E = \perp$), $I_C$ is the set of instances that synchronizes with the condition $C$, $I_{Sub_T}$ is the set of instances that participates in the scenario of $Sub_T$, and $I_{Sub_E}$ is the set of instances that participates in the scenario of $Sub_E$.*

A peculiar case of an *if-then-else* construction is the non-deterministic choice, which uses the reserved word *SELECT* that defines probabilities for the condition to be evaluated as true or false values. Figure 2.12 shows a non-deterministic choice example, which indicates a probability of 50% for the message *Accept* to occur and other 50% for the message *Reject*.

A *loop* construction allows the execution of a scenario several times. LSC language offers three types of loops: *fixed*, *dynamic* and *unbound*.

A *fixed* loop executes a determined number of iterations, depicted at the left corner of the loop's subchart, which can be indicated by a constant or a variable, as shown in Figure 2.13(a).

Alike *fixed* loops, *dynamic* loops have a determined number of iterations, which is defined by the user at execution time in the *Play-Engine* tool. Figure 2.13(b) shows a *dynamic* loop. Such loop is denoted by "?" at the left corner of the loop's subchart.

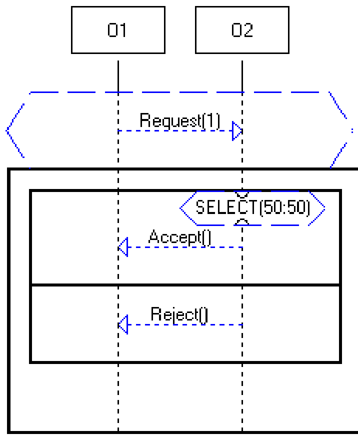


Figure 2.12: A non-deterministic choice

*Unbound* loops, on the other hand, execute infinitely often until a *cold* condition, presented inside of the loop's scenario, is evaluated as false, forcing the loop's scenario to be abandoned, and the execution continues in the next location after the loop. These loops are denoted by "*" at the left corner of the loop's subchart, as shown in Figure 2.13(c).

It is worth pointing out that *dynamic* and *fixed* loops can also be abandoned when a condition inside loop's scenario is evaluated as false.

**Definition 2.3.19.** *(Loop construction) A loop construction* $Loop$ *in a chart* $L$ *is defined as* $Loop = (Kind, \mu, Sub)$*, where* $Kind \in \{Fixed, Unbound, Dynamic\}$ *is the loop's kind,* $\mu \in \mathbb{N} \cup \{\infty\}$ *is the loop's number of iterations, and* $Sub_{LOOP} \in SUB_L$ *is the subchart containing the loop events to be iterated.* $I_{Loop}$ *is the set of instances participating in the loop's scenario.*

LSC language allows to establish time restrictions for real-time systems. The *Play-Engine* tool has a clock (an instance with a property called *time* and a method called *tick*), which is associated to the internal clock of the computer host, so that the time can be manipulated inside of the LSC scenarios. The value of the property *time* informs the current time in time units and the method *tick* increases the current time by a time unit.

A time restriction is basically formed by a combination of assignments and conditions, which can be *cold* or *hot*. Several types of time restrictions can be built inside of a LSC scenario: *Vertical Delay*, *Message Delay* and *Timer*.

*Vertical Delay* indicates a minimum and maximum time allowed between two consecutive events in an instance line. This restriction has an as-

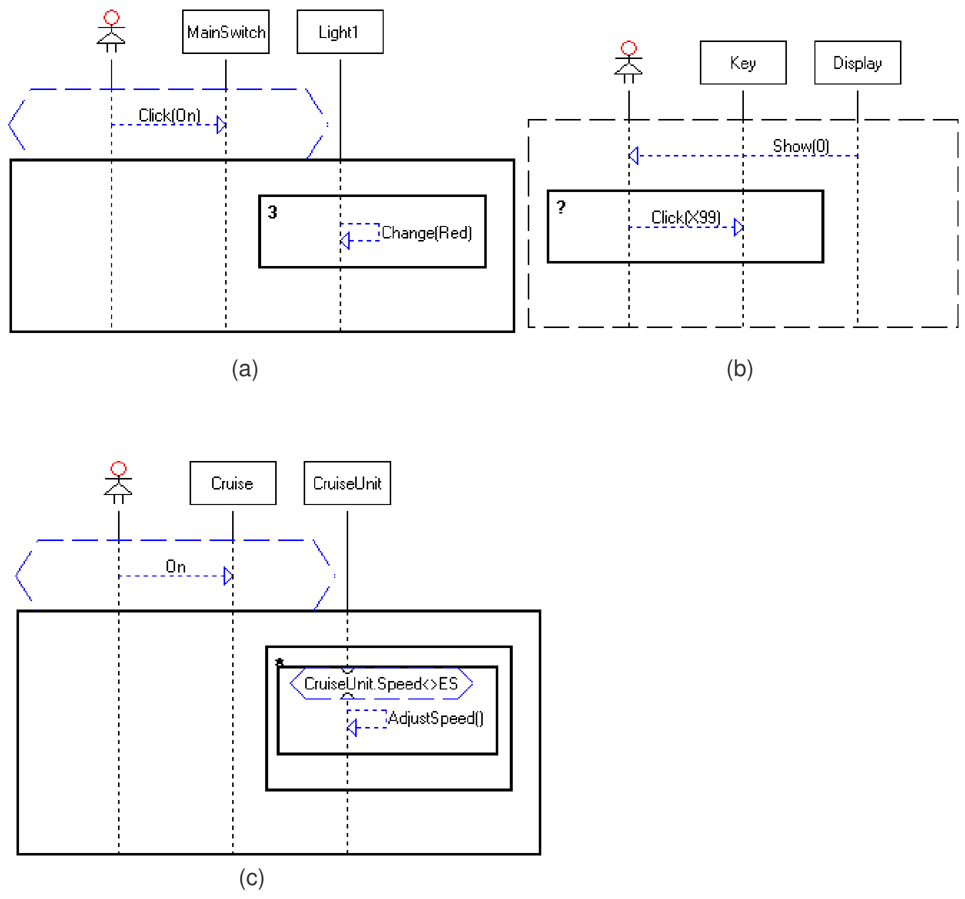


Figure 2.13: *Fixed*, *dynamic* and *unbound* loops

signment and two *hot* conditions. The assignment is used to store the time after the occurrence of the first event. The allowed minimum time is specified with a *hot* condition in the form "*Time Oper Time-Var* + *Min-Delay*", before the second event, where *Time* is the property of *Clock* instance, *Oper* is a relational operator ($>$ or $>=$), *Time-Var* is the variable that stores the time and *Min-Delay* is an integer number. In agreement with the semantic of a *hot* condition (Section 3.6), the execution moves forward when the established period of time have passed. The allowed maximum time is specified with a *hot* condition in the form "*Time Oper Team-Var* + *Max-Delay*", after the second event, where *Time* is the property of *Clock* instance, *Oper* is a relational operator ($<$ or $<=$), *Time-Var* is the variable that stores the time and *Max-Delay* is an integer number. If this condition is reached after the established maximum time has expired, the condition is evaluated to false, causing a requirement violation.

Figure 2.14 shows a *Vertical Delay* time restriction, where message *O1.M1()* must be sent between two and three time units after receiving message *O2.M2()*.

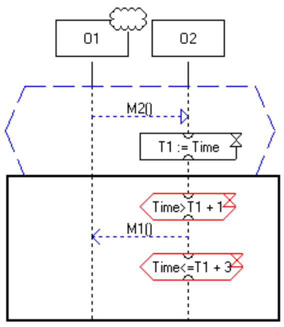


Figure 2.14: A *Vertical Delay* time restriction

*Message Delay* indicates the minimum and maximum delay between sending and receiving a message. This restriction is specified like a *Vertical Delay*, with the exception that the time is stored in an instance line and verified in another instance line, as displayed in Figure 2.15. The scenario of Figure 2.15 specifies that after message *O1.M1()* is sent, it must be received between three and four time units after.

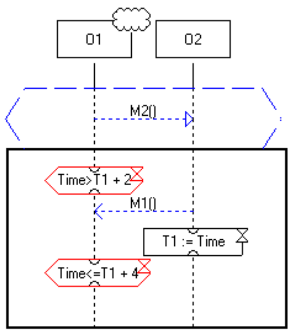


Figure 2.15: A *Message Delay* time restriction

Through *timers*, the LSC language allows to express a minimum time between two consecutive events or a maximum time between two or more consecutive events. Those *timers* cannot be shared by different instances, in other words, just events in the same instance line can be restricted.

A *Timer* is formed starting from a *Vertical Delay* time restriction. Whenever the intention is expressing the minimum delay between two consecutive events, an assignment is used after receiving the first event and a condition is inserted before calling the second event. But if the intention is expressing the maximum delay, the condition should be inserted after calling the second event.

Figure 2.16 shows two examples of *timers*. The first establishes a maximum delay between two consecutive events, in which the event *O1.M1()* and the second call of *O2.M2()* must be executed at most three time units after the first call of *O2.M2()*. The second scenario defines a minimum delay between two consecutive events, in which the event *O1.M1()* must be executed at least two time units after the event *O2.M2()*.

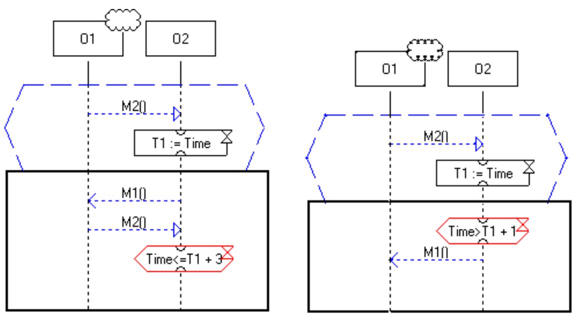


Figure 2.16: *Timers*

## 2.4 Petri Nets (PNs)

PN is a family of formal modeling techniques that allows the modeling of parallel, concurrent, asynchronous and non-deterministic systems.

PNs have an origin dating back to 1962, when Carl Adam Petri [46] wrote his PhD thesis on the subject. Since that time, PNs have been accepted as a powerful formal specification tool. PNs also have applications in a number of different disciplines including engineering, manufacturing, business, chemistry, mathematics, and even within the judicial system.

There are many extensions to PNs. Some of them are completely backwards compatible (e.g. coloured Petri nets) with the original PN, some add properties that cannot be modeled in the original PN (e.g. timed Petri nets). If they can be modeled in the original PN, they are not real extensions, instead are convenient ways of showing the same thing, and can be transformed with mathematical formulas back to the original PN, without loosing any meaning. Extensions that cannot be transformed are sometimes very powerful, but usually lack the amount of mathematical tools available to analyse normal PNs.

The term high-level PN is used for many PN formalisms that extend the basic place/transition one. This includes coloured PNs, hierarchical PNs, and all other extensions sketched below.

In a standard PN, tokens are indistinguishable. In a coloured PN, the values of tokens are typed, and can be tested and manipulated with a functional programming language. A subsidiary of coloured PNs are the well-formed PNs, where the arc and guard expressions are restricted to make it easier to analyse the net.

Another popular extension of PNs is hierarchy, which supports different levels of refinement and abstraction.

Prioritized PNs add priorities to transitions, whereby a transition cannot fire, if a higher-priority transition is enabled (i.e. can fire). Thus, transitions are in priority groups, and e.g. priority group 3 can only fire if all transitions are disabled in groups 1 and 2. Within a priority group, firing is still non-deterministic.

In certain cases, however, the need arises to also model the timing, not only the structure of a model. For these cases, timed PNs have evolved, where there are timed and immediate transitions. A subsidiary of timed petri nets are the stochastic PNs that add non-deterministic time to transitions.

There are other extensions to PNs, however, it is important to keep in mind, that as the complexity of the net increases in terms of extended properties, the harder it is to use standard tools to evaluate certain properties of the net. For this reason, it is a good idea to use the most simple net type possible for a given modeling task.

Place/Transition net (PT-PN, for short called hereafter PN) is the most wide spread PN variant [45, 39]. Its structure consists of nodes, connected by directed segments called arcs. There is two types of nodes, places (P) represented by circles and transitions (T) represented by bars. Arcs connect places to transitions or transitions to places. Figure 2.17 depicts the basic elements of a simple PN.

PN is a multi graph, since it allows multiple arcs from a node to another, it is bi-parted, since the graph elements are parted in two sets (places and transitions) and the arcs connect elements of different groups and it is directed, since the arcs have source and target nodes.

**Definition 2.4.1.** *A PN structure can be formally defined as a quadruple*

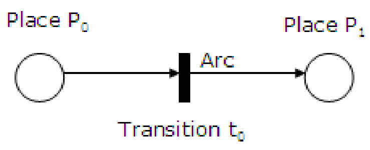


Figure 2.17: PN basic elements

*$(P, T, I, O)$, where:*

- *$P = \{p_1, p_2, ..., p_m\}$ is a set of places, where $m \in \mathbb{N}$ is the number of places in the net;*

- *$T = \{t_1, t_2, ..., t_s\}$ is a set of transitions, where $s \in \mathbb{N}$ is the number of transitions in the net;*

- *$I : P \times T \to N$ is the function that defines the input arcs to the transitions. If $I(p_j, t_i) = k$, then there is $k \in \mathbb{N}$ arcs from place $p_j$ to transition $t_i$, and in the case of $I(p_j, t_i) = 0$, there is no arc from place $p_j$ to transition $t_i$;*

- *$O : T \times P \to N$ is the function that defines the output arcs to the transitions. If $O(t_i, p_q) = k$, then there is $k \in \mathbb{N}$ arcs from transition $t_i$ to place $p_q$, and in the case of $O(t_i, p_q) = 0$, there is no arc from transition $t_i$ to place $p_q$;*

Usually, in the graphic representation, multiple arcs connecting places and transitions are represented in a compact way by a single arc labeling it with its weight or multiplicity $k$ (see Figure 2.18).

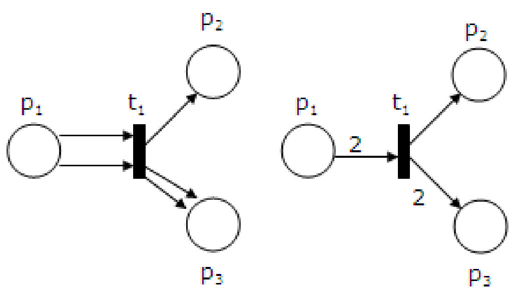


Figure 2.18: Compact representation of a PN

A PN with *tokens* associated to its places is called a marked Petri Net [1] $PN = (P, T, I, O, M_0)$, where $M_0$ is the initial marking. A peculiar distribution $(M)$ of the *tokens* in the places, represents a specific state of the system. These *tokens* are denoted by black dots inside the places.

- $M = (M(p_1), M(p_2), ..., M(p_m))$, where $M(p_j) \in \mathbb{N}$ is the marking of place $p_j$, that is the number of of *tokens* in place $p_j$, and $m \in \mathbb{N}$ is the number of places in the net.

A transition firing represents the occurrence of an event that modifies the state of a system, modifying the current marking $(M_i)$ to a new one $(M_{i+1})$. A transition $t_i \in T$ is said to be enabled to fire if for each input place $p_j$, $I(p_j, t_i) > 0$, the number of *tokens* is at least equal to the arc weight $(I(p_j, t_i))$, so $M(p_j) \geq I(p_j, t_i)$ for any place $p_j \in P$.

A firing of an enabled transition $t_i$ removes, from each input place $p_j$, a number of *tokens* equal to the arc weight $I(p_j, t_i)$ that connects place $p_j$ to transition $t_i$, and adds, to each output place $p_q$, a number of *tokens* equal to the arc weight $O(t_i, p_q)$ that connects the transition $t_i$ to place $p_q$. Figure 2.19 depicts an enabled transition $t_1$ and Figure 2.20 shows the marking after firing this transition.

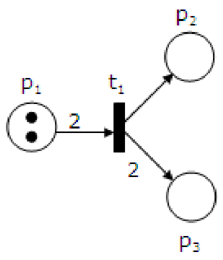


Figure 2.19: Enabled transition $t_1$

The introduction of the concept of inhibitor arc, originally not present in PN, increases the modeling power of PN, adding the ability of testing if a place does not have *tokens* [56]. Figure 2.21 illustrates an inhibitor arc connecting the input place $p_2$ to the transition $t_1$, which is denoted by an arc finished with a small circle. In the presence of an inhibitor arc, a transition is enabled to fire if each input place connected by a normal arc has a number of *tokens* equal to the arc weight, and if each input place connected by an inhibitor arc has no *tokens*. In Figure 2.21, the transition $t_1$ is enabled to fire.

[1]The term PN is adopted for representing both Place/Transition net structure and marked Place/Transition nets whenever is avoided.

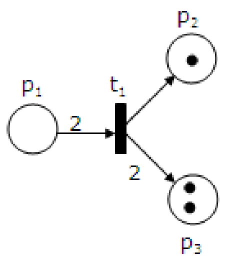


Figure 2.20: Marking after the firing event of transition $t_1$

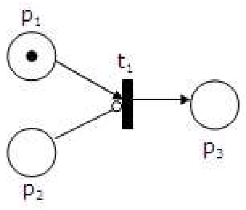


Figure 2.21: PN with an inhibitor arc

Petri Net modeling has some basic net structures from which more complex constructions are accomplished. These basic models are presented below:

**Sequence** represents an execution of an action, since a certain condition is satisfied. After the execution of an action (transition to firing), other action (transition $t_1$) can be fired, since a certain condition ($m(P_1) = 1$) is satisfied (see Figure 2.22).

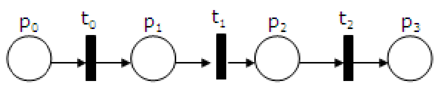


Figure 2.22: Sequence net

**Distribution** is used to create parallel processes starting from parent process. The child processes are created through the distribution of parent's *tokens*. A distribution net is presented in Figure 2.23. It is important to note that if there was a *token* in $p_1$, this *token* would be "propagated" to $p_2$ and $p_3$.

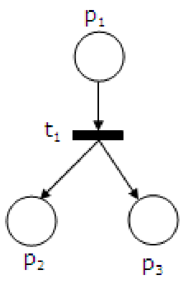


Figure 2.23: Distribution net

**Junction** synchronizes concurrent activities. In Figure 2.24, the transition $t_1$ only fires when $p_1$ and $p_2$ have *tokens*, establishing the synchronism.

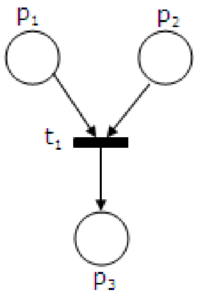


Figure 2.24: Junction net

**Non-deterministic Choice** is specified by a set of conflicting transitions, where choosing which should fire is carried out in non-deterministic manner (see Figure 2.25). The conflict can be classified as structural or effective. Both conflicts are associated to the fact of two transitions possess the same set of places as input. However, if the net does not have *tokens*, the conflict is said to be structural. If there is a single *token* in the common input place to the transitions, the conflict is said to be effective. Figure 2.26 illustrates both conflicts.

The study of PN properties allows a detailed analysis of the modeled system. Two types of PN properties can be distinguished:

**Behavioral Properties** depend on both the state (or on initial marking) and on PN's structure;

**Structural Properties** depend on PN's structure.

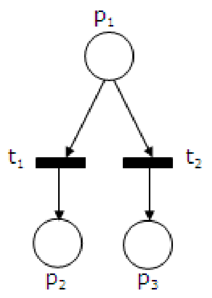


Figure 2.25: Non-deterministic Choice net

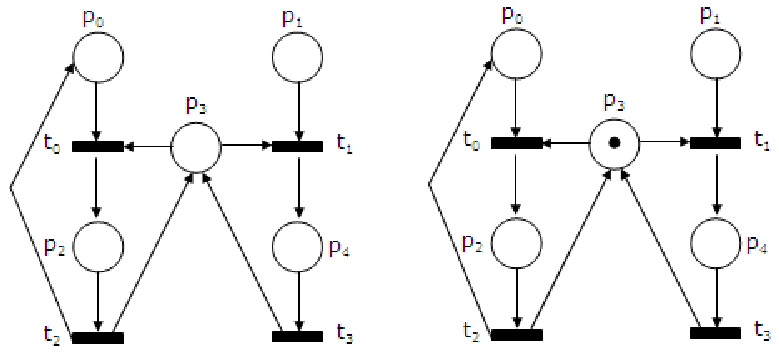


Figure 2.26: Structural and Effective conflicts

Among behavioral properties, we can highlight:

**Reachability** refers to the possibility of a system to reach a certain state. A marking $M_i$ is reachable starting from marking $M_0$, if there is a transition firing sequence that takes the net with marking $M_0$ to the marking $M_i$. A firing or occurrence sequence is denoted by $\sigma = M_0\ t_1\ M_1\ t_2\ M_2\ ...\ t_i\ M_i$ or simply $\sigma = t_1\ t_2\ ...\ t_i$. In this case, $M_i$ is reachable from $M_0$ by $\sigma$, so $M_0[\sigma > M_i$. The set of all possible reachable markings from $M_0$ in a net $(N, M_0)$ is denoted by $R(N, M_0)$ or simply $R(M_0)$. The set of all possible firing sequences from $M_0$ in a net $(N, M_0)$ is denoted by $L(N, M_0)$ or simply $L(M_0)$;

**Boundedness** A PN is said to be *k-bounded* if the number of *tokens* in each place does not exceed a finite number $k$ for any reachable marking from $M_0$, i.e., $M_p \leq k$ for every place $p$ and every marking $M \in R(M_0)$. A PN is said to be *safe* if it is *1-bounded*. Figure 2.27 depicts a *bounded* PN and Figure 2.28 depicts a *safe* PN;

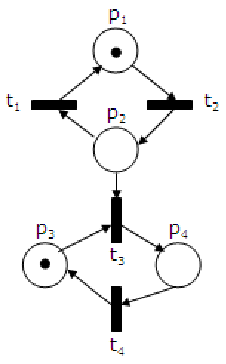


Figure 2.27: A *bounded* PN

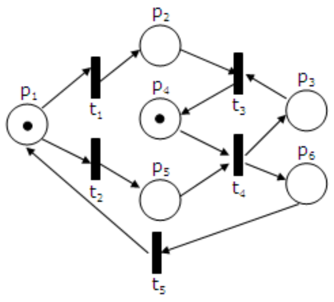


Figure 2.28: A *safe* PN

**Deadlock Freedom** A PN is said to be deadlock free if there is no reachable marking such that no transition is enabled;

**Liveness** A PN $(N, M_0)$ is said to be *live* if, no matter what marking has been reached from $M_0$, it is possible to fire any transition of the net by progressing through some further firing sequence. This means that a *live* PN guarantees deadlock-free operation, no matter what firing sequence is chosen. Figure 2.29 depicts a *live* PN. However, a deadlock free PN might not be *live*, if a transition does not belong to a firable transition sequence in any reachable marking;

**Reversibility** A PN is said to be *reversible* if, for each marking $M \in R(M_0)$, $M_0$ is reachable from $M$. Thus, in a *reversible* net, one can always get back to the initial state. In many applications, it is not necessary to get back to the initial state as long as one can get back to some (*home*) state. A marking $M^{'}$ is said to be a *home* state if, for each marking $M \in R(M_0)$, $M^{'}$ is reachable from $M$. Figure 2.30 depicts a

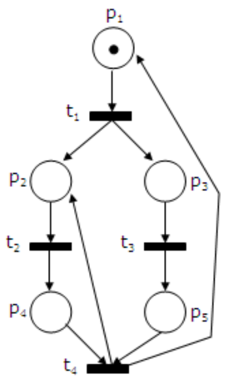


Figure 2.29: A *live* PN

*reversible* PN;

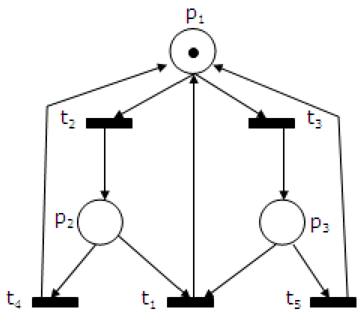


Figure 2.30: A *reversible* PN

**Coverability** A marking $M$ in a PN $(N, M_0)$ is said to be *coverable* if there is a marking $M' \in R(M_0)$ such that $M'(p) \geq M(p)$ for each $p$ in the net;

**Persistence** A PN $(N, M_0)$ is said to be *persistent* if, for any two enabled transitions, the firing of one transition will not disable the other. A transition in a *persistent* net, once it is enabled, will stay enabled until it fires. The net presented in Figure 2.29 is *persistent*;

**Fairness** Many different notions of fairness have been proposed in the literature on PN. We present here two basic fairness concepts: *bounded-fairness* and *unconditional fairness*. Two transitions $t_1$ and $t_2$ are said to be in a *bounded-fair*(*B-fair*) relation if the maximum number of times that either one can fire while the other is not firing is *bounded*. A PN $(N, M_0)$ is said to be a *B-fair* net if every pair of transitions in the net are in *B-fair* relation. A firing sequence $\sigma$ is said to be *unconditionally fair* if it is finite or every transition in the net appears infinitely often in $\sigma$. A PN $(N, M_0)$ is said to be an *unconditionally fair* net if every firing sequence $\sigma$ from $M \in R(M_0)$ is *unconditionally fair*. Figure 2.31 depicts a *B-fair* PN, while the net presented in Figure 2.29 is *unconditionally fair*.

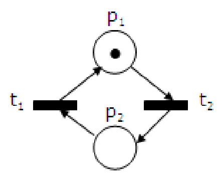


Figure 2.31: A *B-fair* PN

Structural properties, include:

**Structural Liveness** A PN $N$ is said to be *structurally live* if there is a *live* initial marking for $N$. Figure 2.32 depicts a *structurally live* PN;

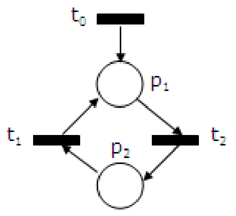


Figure 2.32: A *structurally live* PN

**Structural Boundedness** A PN $N$ is said to be *structurally bounded* if it is bounded for any finite initial marking $M_0$. Figure 2.33 depicts a *structurally bounded* PN;

**Conservativeness** A PN $N$ is said to be *conservative* if there is a positive integer $y$ for every place $p$ such that the weighted sum of *tokens*, $M^T Y = M_0^T Y =$ a constant, where $Y = [y_i], y_i \in \mathbb{N}$, for every $M \in R(M_0)$ and for any fixed initial marking $M_0$. Figure 2.34 depicts a *conservative* PN;

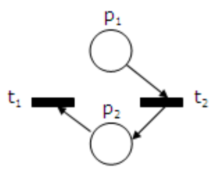


Figure 2.33: A *structurally bounded* PN

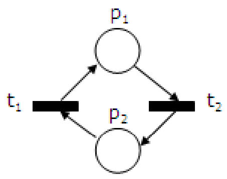


Figure 2.34: A *conservative* and *consistent* PN

**Repetitiveness** A PN is said to be *repetitive* if there is a marking $M_0$ and a firing sequence $\sigma$ from $M_0$ such that every transition occurs infinitely often in $\sigma$. Figure 2.35 depicts a *repetitive* PN;

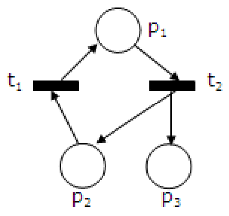


Figure 2.35: A *repetitive* and *consistent* PN

**Consistency** A PN is said to be *consistent* if there is a marking $M_0$ and a firing sequence $\sigma$ from $M_0$ back to $M_0$ such that every transition occurs at least once in $\sigma$. The nets presented in Figure 2.34 and Figure 2.35 are *consistent*

PNs can be grouped in two classes: Ordinary and Non-Ordinary (high level). Ordinary nets use a basic type of *tokens*, the non-negative integer type. Making an analogy to programming languages, high level nets [19] can possess more sophisticated *tokens*, as data types defined by the user or composed types that are formed by several elementary types. The ordinary nets are subdivided in:

**Condition event and Elementary nets** are the most elementary [48, 49]. This kind of net allows, at most, one *token* in each place and all arcs have an unitary value;

**Place-Transition net** allows the accumulation of *tokens* in the same place, as well as natural numbers value in its arcs.

The high level nets differ from the ordinary ones, because they individualize the *tokens*. This individualization can be realized through several artifices, for instance, *tokens's* color or objects representing the *tokens*. High level nets allows a higher modeling abstraction. Coloured Petri Nets (CPNs) [28], presented in the next section, is an example of a high level PN.

## 2.5 Coloured Petri Nets (CPNs)

Coloured Petri Nets (CPNs) [29, 28] use the power of the programming languages providing compact descriptions of concurrent systems by including abstract data types within the basic Petri net framework.

Each place in a CPN model has an associated type which is defined in a set of declarations in a language called CPN ML, a variant of Standard ML [44].

A marking of a place defines a collection of data values, known as *tokens*, that are associated with that place. The *token's* value ranges over the type of the place. This collection of *tokens* is a multi-set, since it may contain several *tokens* of the same value. CPNs also include the initial state of the system, called the initial marking.

Transitions in a CPN model may also have guards associated with them, which are included in square brackets. Guards are boolean expressions which are important for describing CPN dynamics.

Arcs in a CPN model have expressions associated with them. The expressions are built from constants, variables and functions and are written next to their associated arcs using CPN ML language. The functions are defined, and constants and variables declared, in a set of CPN ML declarations.

Inscriptions are associated to CPN net components, i.e. places, arcs, and transitions. Some inscriptions are CPN ML constructs that affect the behavior of a net, while other inscriptions do not affect the behavior of nets. Inscriptions vary in their syntactic requirements depending on the type of inscriptions. There are three types of inscriptions: *place inscriptions*, *arc inscriptions* and *transition inscriptions*.

There are three inscriptions that may be associated with a place. Two are optional and one is required:

**Colour set inscription** It determines the colour set, i.e. the type, of all places;

**Initial marking inscription** It is a multi-set expression that specifies the initial tokens for each place. The initial marking inscription is optional;

**Place name inscription** It is an optional label that identifies the place, and it may contain any sequence of characters.

Arcs have only one inscription – the arc inscription. An arc inscription is a CPN ML expression that evaluates to a multi-set or a single element.

There are four inscriptions that may be associated with a transition. All are optional:

**Transition name inscription** It is an optional label that identifies the transition, and it may contain any sequence of characters;

**Guard inscription** A guard is a CPN ML boolean expression that evaluates to true or false;

**Time inscription** A transition delay must be a positive integer expression. The expression is preceded by @+, and this means that the time inscription has the form @+ *delay-expr*;

**Code segment inscription** Each transition may have an attached code segment which contains ML code. Code segments are executed when their parent transition occurs.

Types, variables and functions are defined in what is called the declarations of a CPN. They are written in the functional programming language ML [44]. The variant, known as CPN ML, has some special key words. *Color* is used to denote a type.

Types can be simple colour sets as *boolean*, *integer*, *string* and *enumerated*, as well as they can be compound colour sets as *record*, *list* and *union*. Besides, types can be *timed*. A colour set is timed by appending the keyword *timed* to the end of its declaration.

CPN variables are used in CPN inscriptions. *Binding* is the association of a value with a variable. A binding has both scope and content. *Scope* is the locations in a model in which a particular binding can be referred. *Extent* is the interval during which a particular binding is in effect. A CPN variable has the following characteristics:

- they are declared using the reserved word *var* and the name of a previously declared colour set;

- they are bound to a variety of different values (from their colour set) when evaluates if a transition is enabled;

- a variable is bound to a value, the scope of a variable is local to the transition;

- there can be multiple bindings simultaneously active on different transitions. These bindings can exist simultaneously because they have different scopes;

- the extent of a CPN variable binding is the firing of a particular transition;

- they provide arc inscriptions with the ability to refer different values.

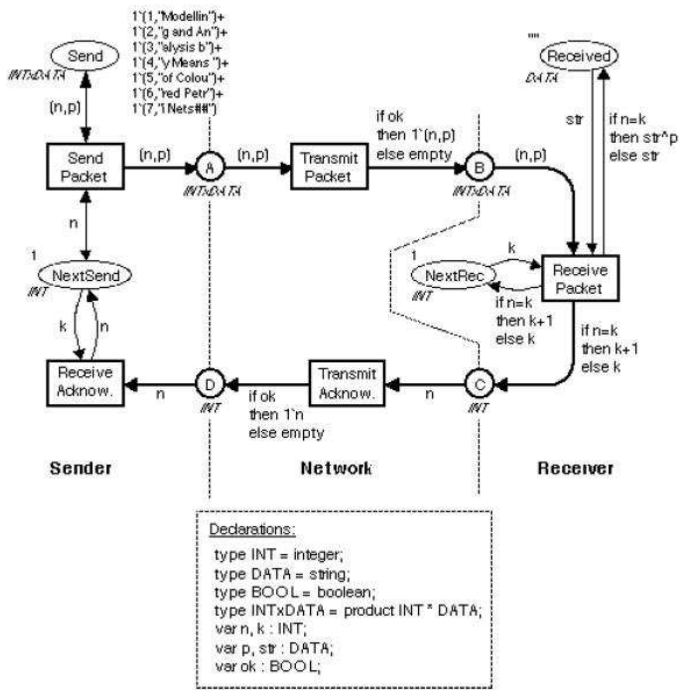
The CPN ML identifiers are alphanumeric sequences of letters, digits, primes/apostrophes ('), and underscores (_) – starting with a letter. They are used for: colour sets, record colour set field labels, value constructors, variables, operators and function symbols, prefixes of place, transition, and page names.

Transitions can be enabled and can then occur (fire). A transition is enabled if its input places have the required *tokens* and its guard is evaluated as true. These enabling requirements are determined by binding the transition's variables to values taken from their types. The required *tokens* are defined by evaluating the input arc expressions for a particular binding of the variables. The same binding is used for evaluating the guard. The occurrence of a transition removes tokens from its input places and adds *tokens* to its output places. The removed tokens are defined by the evaluated expressions on the corresponding incoming arcs for this binding of variables, while the values of the added tokens are determined by evaluating the arc expressions on the corresponding outgoing arcs for the same binding. Hence, transitions can occur in different modes, depending on the bindings of the variables.

CPN models can be created using hierarchical constructs. Hierarchies are built using the notion of a substitution transition, which may be considered a macro expansion. The model starts with a top-level CPN diagram, which provides an overview of the system being modeled and its environment. In hierarchical CPNs, the top-level diagram contains a number of substitution transitions. Each of these substitution transitions is then refined by another CPN diagram, which may also contain substitution transitions. The top-level diagram and each of the substitution transitions are defined by a module, called a page. The relationships between the different pages are defined by a page hierarchy. The page hierarchy also includes the name of the page that defines the declarations required for the CPN inscriptions, called the Global Declaration node.

**Definition 2.5.1.** *A CPN model is a nine-tuple $C_{NET} = (\Sigma, P, T, A, N, C, G, E, I)$, where $\Sigma$ is a finite set of non-empty types, called color sets, $P$ is a finite set of places, $T$ is a finite set of transitions, $A$ is a finite set of arcs, $N : A \rightarrow P \times T \cup T \times P$ is a node function, $C : P \rightarrow \Sigma$ is a color function, $G$ is a guard function. It is defined from $T$ into expressions such that $\forall t \in T : [Tp(G(t)) = Bool \wedge Tp(Var(G(t))) \subseteq \Sigma]$, $E$ is an arc function. It is defined from $A$ into expressions such that $\forall a \in A : [Tp(E(a)) = C(p(a))_{MS} \wedge Tp(Var(E(a))) \subseteq \Sigma]$, where $p(a)$ is the place of $N(a)$ and $C_{MS}$ denotes the set of all multi-sets over $C$, $I$ is an initialization function. It is defined from $P$ into expressions such that $\forall p \in P : [Tp(I(p)) = C(p)_{MS} \wedge Var(I(p)) = \oslash]$, where $Tp(expr)$ denotes the type of an expression, $Var(expr)$ denotes the set of variables in an expression, $C(p)_{MS}$ denotes a multi-set over $C(p)$.*

A small example [1] of a CPN is shown in Figure 2.36. It describes a simple transport protocol transferring a number of packets over a unreliable network from a sender to a receiver. The ellipses and circles are called places. They describe the local states of the system. The rectangles are called transitions. They describe the actions. The arrows are called arcs. The arc expressions describe how the state of the CPN changes when the transitions occur. Each place contains a set of marks called *tokens*. In contrast to low-level PNs (such as Place/Transition Nets), each of these tokens carries a data value, which belongs to a given type. As an example, place *Send* has seven *tokens* in the initial state. All the *token* values belong to the type *INTxDATA* and they represent seven packets which are ready to be sent. Each of the places *NextSend* and *NextRec* starts with a single token with value 1 (belonging to type *INT*). Place *Received* starts with a token which contains the empty string “” (belonging to type *DATA*). To be able to occur, a transition must have sufficient *tokens* on its input places, and these *tokens* must have token values that match the arc expressions. As an example, let us consider transition *SendPacket*. It has three surrounding arcs of which two are double arcs. The three arc expressions involve the variable *n* of type *INT* and the variable *p* of type *DATA*. In order to fire transition *SendPacket*, we must bind these two variables to values in their types, in such a way that the arc expression of each incoming arc evaluates to a *token* value that is present on the corresponding input place. Since *NextSend* only contains one *token* with value 1, it is obvious that *n* must be bound to 1. Next we see that *p* must be bound to *"Modellin"*, since *Send* only has one *token* in which the first element of the pair is 1. With the binding <*n=1, p="Modellin"*> transition *SendPacket* is enabled, because there is a 1 *token* on place *NextSend* and a *(1,"Modellin") token* on place *Send*. When the transition occurs, it removes the two specified *tokens* from the input places, but it immediately puts two other with the same values back, due to the two double arcs. Simultaneously, it produces a copy of the *(1,"Modellin") token* on place *A*. When the *(1,"Modellin") token* is put on place *A*, transition *TransmitPacket* becomes enabled with two different bindings: <*n=1, p="Modellin", ok=true*> and <*n=1, p="Modellin", ok=false*>. If the first binding is chosen, the packet is transferred from place *A* to place *B*. If the second binding is chosen, the packet is lost on the network.



```
Declarations:
 type INT = integer;
 type DATA = string;
 type BOOL = boolean;
 type INTxDATA = product INT * DATA;
 var n, k : INT;
 var p, str : DATA;
 var ok : BOOL;
```

Figure 2.36: A Simple CPN

CPNs are used to three different purposes.

First of all, a CPN model is a description of the modeled system, and it can be used as a specification (of a system to be built) or as a presentation of a system to be explained to other people, or ourselves. By creating a model, we can investigate a new system before we construct it. This is an obvious advantage, in particular for systems where design errors may harm security or be expensive to correct.

Secondly, the behavior of a CPN model can be evaluated, either by simulation (which is equivalent to program execution and program debugging) or by means of more formal analysis methods (which are equivalent to program verification).

Finally, it should be understood that the process of creating the description and performing the analysis usually gives the modeler a dramatically improved understanding of the modeled system – and it is often the case that this is more valid than the description and the analysis results themselves.

CPN models can be evaluated in many different ways, similar to basic Petri Nets, in which the same properties can be analysed and verified, but using different techniques.

The first method is the interactive simulation. It is very similar to debugging and prototyping. This means that we can execute a CPN model, to make a detailed investigation of the behavior of the modeled system.

The second method is the automatic simulation which is similar to program execution. It allows a fast execution of thousands or millions of transitions. The purpose is to investigate the functional correctness of the system or to investigate the performance of the system, e.g. to identify bottlenecks, to predict the use of buffer space or the mean/maximal service time.

The third approach is based on the analysis of occurrence graphs (also called state spaces or reachability graphs). The basic idea behind occurrence graphs is the construction of a directed graph which has a node for each reachable system state and an arc for each possible state change. Obviously, such a graph may become very large, even for small CPNs. However, it can be constructed and analysed totally automatically, and there exist techniques which makes it possible to work with condensed occurrence graphs without losing analytic power.

The fourth method is based on place invariants. This method is very similar to the use of invariants in ordinary program verification. The user constructs a set of equations which is verified to be satisfied for all reachable system states. The equations are used to verify properties of the modeled system, e.g., absence of deadlock.

## 2.6 Concluding Remarks

This chapter presented some details on object-oriented analysis and design, and described a formal approach to properties verification.

MSCs and UML sequence diagram are a popular mechanisms for specifying scenarios that describe possible interactions between processes or objects. However these specification languages possess some disadvantages: they can not specify what must occur for all system run and they can not specify anti-scenarios.

LSC is a system specification language based on scenarios that allows to specify anti-scenarios as well as it permits specify what should happen for all system runs. However, analysis and verification is not possible.

As the mentioned specification languages do not handle properties analysis and verification, a formal approach should be used in order to provide such tasks, hence permitting to verify some properties at the beginning of the project contributing to reduce some risks that may lead to project failure.

PN is a family of formal modeling techniques that allows the modeling of parallel, concurrent, asynchronous and non-deterministic systems. They are presented as a possible approach to formal verification.

CPN is a class of PNs that use the power of the programming languages providing compact descriptions of concurrent systems by including abstract data types within the basic Petri net framework.

# Chapter 3

# Mapping LSC to CPN

*This chapter describes how to obtain the corresponding CPN models for the LSC constructions presented in the previous Chapter. Once the individual models were obtained, a joining process composes those individual models and provides a final model that represents the LSC scenarios. Finally, the semantics of LSC and CPN models are compared.*

## 3.1 Object Properties, Types and Variables

In the LSC language, the system is modeled using object oriented notions and terminologies, in which, a system is composed of objects that represent instances of a given class, which are formed by properties based on some application data type that can be *Enumerated*, *Discrete*, *String*.

In order to obtain a CPN type that represents the data type used in an LSC specification, the following steps should be followed:

1. for mapping an enumerated type, it is necessary to create a CPN type declared like *color enum-name = with* $id1|...|idn$, where *color* and *with* are CPN ML reserved words, *enum-name* is the type name and $id1|id2|...|idn$ are items of the enumerated set. For example, *color week = with* $Sun|Mon|Tue|Wed|Thu|Fri|Sat$;

2. for mapping a discrete type, a CPN type like *color type-name = int with* $min..max$ or *color type-name = real with* $min..max$ should be created, where *type-name* is the type's name, *int* and *real* are CPN ML primitive types. For example, *color byte = int with 0..127*;

3. The String type is represented by *color type-name = string*, where

*type-name* is the type's name, *string* is a CPN ML primitive type. For example, *color name = string*.

**Definition 3.1.1.** *(Function for mapping application types) Let $AT$ be the set of LSC application types (domains), $D \in AT$ be a finite set of values, and $\Sigma$ be a finite set of non-empty CPN types. A function $M_{DT} : D \mapsto \Sigma$ that maps an LSC type to a CPN type must be defined as:*

- *$M_{DT}(d) =$ "color $D$ = with $id_1|...|id_i$", $id_i \in D$, if $D$ is an enumerated type;*
- *$M_{DT}(d) =$ "color $D$ = int with $min..max$", if $D = \langle d_{min}, d_i, d_{max} | d_{min}, d_i, d_{max} \in \mathbb{N}\rangle$ is an ordered set of values, $|D| = n \in \mathbb{N}$, and $D$ is a discrete type;*
- *$M_{DT}(d) =$ "color $D$ = real with $min..max$", if $D = \langle d_{min}, d_i, d_{max} | d_{min}, d_i, d_{max} \in \mathbb{Q}\rangle$ is an ordered set of values, $|D| = n \in \mathbb{N}$, and $D$ is a discrete type;*
- *$M_{DT}(d) =$ "color $D$ = string", if $D$ is a string type.*

A LSC object has several properties which are of a certain type. In order to represent an LSC object in a CPN manner, it must be declared a CPN type which encapsulates object's properties, as "*color TypeName=record id1:Type1*...*idn:Typen*", where *color* and *record* are CPN ML reserved words, *TypeName* is the object name, *id1...idn* are the properties names, "*" is a symbol used to separate property's definition and *Type1...Typen* are the properties types which must be created following the rules for mapping a LSC application data type.

**Definition 3.1.2.** *(Function for representing an object) Let $O$ be the set of concrete objects, $o_i \in O$ be a concrete object of some class. A function $M_{DTI} : O \mapsto \Sigma$ that maps each LSC object into a CPN type which represents their structures must be defined as: $M_{DTI}(o_i) =$ color $o_i.Name =$ record $o_i.C.p_1.Name : M_{DT}(o_i.C.p_1.D) * ... * o_i.C.p_j.Name : M_{DT}(o_i.C.p_j.D)$, where $o_i.C$ is the object's class, $o_i.Name$ is the object's name, $o_i.C.p_j$ is a property of the object $o_i$, $o_i.C.p_j.Name$ is the name of property $p_i$ of the object $o_i$ and $o_i.C.p_j.D$ is the domain of property $p_i$.*

In order to enable the modeling of more general scenarios, the LSC language allows the use of variables instead of using constant values. The variable's type is one of the application data types and its value is picked up from the type's domain.

**Definition 3.1.3.** *(Variable) Let $AT$ be the set of LSC application types, $V_L = \{v_i\}$ be the set of variables of a chart $L$, $Type : V_L \mapsto AT$ be a function that returns the variable's type. A variable $v_i \in V_L$ is represented by its value, so $v_i$ is a value within the domain $Type(v_i)$.*

**Definition 3.1.4.** *(Function for mapping an LSC variable) Let $VS = \{vs_i\}$ be a set of CPN variables and $v_i \in V_L$ be an LSC variable. A function $MV : V_L \to VS$ maps each LSC variable to CPN variable, where $vs_i$ should be declared as: $var\ Nm(L) \oplus$ "_" $\oplus\ v_i : M_{DT}(v_i)$, where $Nm$ is a function that returns the chart's name and $\oplus$ represents a concatenation operation.*

After mapping LSC application data types, it is necessary to create CPN variables, based on CPN created types in order to represent the instances that are used in LSC charts:

- if the instance is the *User* or the *Environment*, that are mere actors and do not have properties, the variable should be of *int* type, and the names should be composed of the chart's name, followed by "_User" or "_Env" depending on the instance;

- for other instances, the variable should be of type that represents the structure of the object. The name of the variable is formed by the chart's name, followed by "_", accompanied by the instance's name.

**Definition 3.1.5.** *(Function for mapping LSC instances) Let $L$ be an LSC chart, $I_L = \{i_i\}$ be the set of instances of $L$, $i_i.O$ be the concrete object of instance $i_i$, $i_i.O.Name$ be the name of concrete object $i_i.O$, $VS$ be the set of CPN variables and $vs_i \in VS$ be a CPN variable. A function $M_O : I_L \mapsto VS$ maps each LSC instance to a CPN variable, where $vs_i$ should be declared as: $var\ vs_i : M_{DTI}(i_i.O)$, where $var$ is a CPN ML reserved word. The label of $vs_i$ must be created in the following way:*

- $Nm(L) \oplus$ *"_User", if* $i_i = User$;

- $Nm(L) \oplus$ *"_Env", if* $i_i = Env$;

- $Nm(L) \oplus i_i.O.Name$, *if* $i_i \neq User \wedge i_i \neq Env$.

## 3.2 Charts

The beginning and ending of prechart and chart body of a scenario are considered synchronization points for the participant instances. Every participant instance of the prechart enter in this scenario simultaneously, while the chart body can only be reached after all these instances have executed their activities successfully. The CPN models which represent the synchronization points at the beginning and ending of prechart and chart body are described in the following way:

1. observe each instance separately and for each synchronization point this instance participates, create a transition, whose label is defined by the following rules:
   - "*chart-name*_Pch_Start" to the beginning of prechart;
   - "*chart-name*_Pch_End" to the ending of prechart;
   - "*chart-name*_CB_Start" to the beginning of chart body;
   - "*chart-name*_CB_End" to the ending of chart body,
     where *chart-name* is the name of the diagram.
2. create one input and one output place for each transition created in the previous step. These places should be of the type that was created to represent the instance;
3. the CPN variable that represents the instance should be assigned to the inscriptions of the input and output arcs of the transition created in Step 1.

When mapping the events inside of a chart, the mapping process takes each event from up to bottom and generates their corresponding CPN models. These CPN models are joined in order to obtain a final CPN model, which represents the whole specification. So, $N$ can be defined as the set composed of all individuals CPN models. $N_i \in N$ is defined as a tuple ($\Sigma$, $P$, $T$, $A$, $C$, $G$, $E$, $I$), where $\Sigma$ is a finite set of non-empty types, which is formed by applying the steps presented earlier, $P$ is the set of places, $T$ is the set of transitions, $A$ is a set of arcs, $C$ is a color function that assigns a color to a place, $G$ is a guard function, $E$ is an arc function that assigns inscriptions to arcs and $I$ is an initialization function, which picks up a random value for each type used in the CPN model. When describing the mapping process, if an element of this tuple is not presented in the formed net structure, it is because there is no need to specify this element at that point.

Let $t_i \in T$ be a transition, we denote ${}^\bullet t_i$ as the set of inputs to transition $t_i$ and $t_i^\bullet$ as the set of outputs to transition $t_i$.

When mapping the synchronization points at the beginning and ending of a prechart and chart body, the mapping process must be applied to each instance participating in each corresponding scenario (prechart or chart body).

**Definition 3.2.1.** *(Function for mapping synchronization points at charts) Let $i_i \in I_L$ be an instance of chart $L$, $l_i \in l_L$ be a location of chart $L$ and $LAB$ be a function that assigns a label to a transition. A function $SP : l_L \mapsto N$, that maps each location that represents a synchronization point at the beginning or ending of a prechart or a chart body to a CPN structure, must be applied, where $l_i \in loc([Pch, Start]) \vee l_i \in loc([Pch, End]) \vee l_i \in loc([CB, Start]) \vee l_i \in loc([CB, End])$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(i_i)$;
- $LAB(t_{ii}) = \begin{cases} Nm(L) \oplus \text{“\_Pch\_Start”}, & \text{iff } l_i \in loc([Pch, Start]); \\ Nm(L) \oplus \text{“\_Pch\_End”}, & \text{iff } l_i \in loc([Pch, End]); \\ Nm(L) \oplus \text{“\_CB\_Start”}, & \text{iff } l_i \in loc([CB, Start]); \\ Nm(L) \oplus \text{“\_CB\_End”}, & \text{iff } l_i \in loc([CB, End]). \end{cases}$

Figure 3.1 shows how the CPN model for *MainSwitch* instance, presented in Figure 2.3, was obtained by applying the rules for mapping the synchronization points. Figure 2.3 shows that *MainSwitch* instance participates in prechart and chart body sections, therefore four transitions must be created, according to the presented steps.

Inside a LSC chart, it is possible to delimit scenarios using *subcharts*.

Alike prechart and chart body, the beginning and ending of a subchart are synchronization points. In order to mapping the synchronization points of a subchart, the following steps must be followed:

1. observe each instance and create a transition for each synchronization point this instance participates, whose label is defined by the following rules:

   - “*chart-name*_Sub_*ID*_Start” to the beginning of subchart;
   - “*chart-name*_Sub_*ID*_End” to the ending of subchart,

   where *chart-name* is the chart’s name and *ID* is an unique sequential number greater than zero.

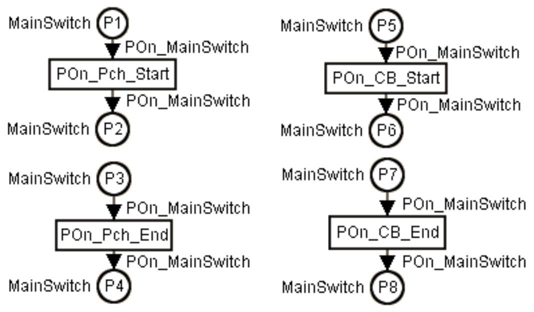


Figure 3.1: Representation of synchronization points for the *MainSwitch* instance

2. create one input and one output places for each transition created in the previous step. These places should be of the type that was created to represent the instance;

3. the CPN variable that represents the instance should be assigned to the inscriptions of the input and output arcs of the transition created in Step 1.

**Definition 3.2.2.** *(Function for mapping synchronization points at subcharts) Let $Sub \in SUB_L$ be a subchart in chart $L$, and $\mathbb{N}^+$ be the set of natural numbers excluding zero. A function $SPS : l_L \mapsto N$, that maps each location that represents a synchronization point at the beginning or end of a subchart to a CPN structure, must be applied, where $l_i \in loc([Sub, Start]) \vee l_i \in loc([Sub, End])$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(i_i)$;
- $LAB(t_{ii}) = \begin{cases} Nm(L) \oplus \text{“\_Sub\_ID\_Start”}, & \text{iff } l_i \in loc([Sub, Start]); \\ Nm(L) \oplus \text{“\_Sub\_ID\_End”}, & \text{iff } l_i \in loc([Sub, End]) \end{cases}$,

  *where $ID \in \mathbb{N}^+$.*

Figure 3.2 shows how the synchronization points of the subchart presented in Figure 2.5 are mapped.

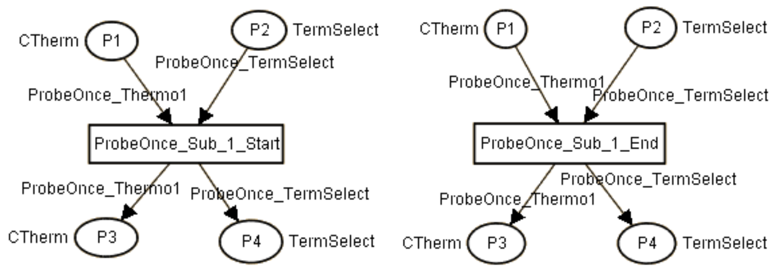


Figure 3.2: Representation of synchronization points for the *Thermo1* instance

## 3.3 Messages

A message may represent a minimum event, i.e. event responsible for enabling a chart. The following steps should be taken to obtain a CPN model of a message that is a minimum event and modifies a property value of some instance.

For the sender instance, it should:

1. create a transition, whose label is formed by the name of the receiver instance followed by "_" accompanied by the property's name that is being modified, followed by "_", accompanied by the value that is being assigned to the property;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;

3. assign the CPN variable that identifies the instance to the inscriptions of the input and output arcs of the transition.

**Definition 3.3.1.** *(Function for mapping a minimum event for sender instance) Let $l_i \in l_L$ be a location of chart L, $m_i \in M_L$ be an LSC message, $m_i.i_{Src}$ is the instance sending the message, $m_i.i_{Dst}$ is the instance receiving the message, $m_i.i_{Dst}.O$ is the concrete object that represents the instance $m_i.i_{Dst}$, $m_i.i_{Src}.O$ is the concrete object that represents the instance $m_i.i_{Src}$, $m_i.i_{Dst}.O.Name$ is the name of the concrete object of receiving instance, $m_i.m_S.P.Name$ is the name of the property been altered, and $m_i.m_S.V$ is the value assigned to the property $m_i.m_S.P$. A function $ME_S : l_L \mapsto N$ maps an instance's localization for sending event of a minimum event to a CPN structure, must be applied, where $l_i \in loc([M_i, Send])$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Src}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Src})$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ "_" $\oplus\, m_i.m_S.P.Name \oplus$ "_" $\oplus\, m_i.m_S.V$.

For the receiver instance, it should:

1. create a transition, whose label is formed by the name of the receiver instance, followed by "_", accompanied by the property's name that is being modified, followed by "_", accompanied by the value that is being assigned to the property;
2. create an input and an output places for the transition created in the previous step. The type of these places should be the type created to represent the receiver instance;
3. assign the CPN variable that identifies the instance to the inscription of the input arc of the transition;
4. assign the expression "*cs.set_idi c v*" to the inscription of the output arc of the transition, where *cs* is the type that represents the instance, *idi* is the property's name that is being modified, *c* is the variable that represents the instance and *v* is the value that is being assigned to the property.

**Definition 3.3.2.** *(Function for mapping a minimum event for receiver instance) A function $ME_R : l_L \mapsto N$, that maps an instance's localization for receiving event of a minimum event to a CPN structure, must be applied, where $l_i \in loc([m_i, Rcv])$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Dst}.O)$;
- $E((p_{ii}, t_{ii})) = M_O(m_i.i_{Dst})$;

- $E((t_{ii}, p_{ii+1})) = M_{DTI}(m_i.i_{Dst}.O) \oplus$ *"set_"* $\oplus\ m_i.m_S.P.Name \oplus M_O(m_i.i_{Dst}) \oplus$ *" "* $\oplus\ m_i.m_S.V$;

- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ *"_"* $\oplus\ m_i.m_S.P.Name \oplus$ *"_"* $\oplus\ m_i.m_S.V$.

In Figure 2.3, the message *Click(On)* is a minimum event, so by applying the above steps for the sender and receiver instances, we obtain the corresponding CPN models shown in Figure 3.3(a) and Figure 3.3(b), respectively. The transition *MainSwitch_Power_On* is created by applying the first step for sender instance and the corresponding input and output places are created following the second step for the same instance. The variable *POn_User* that represents the *User* must be assigned to the arc inscriptions of those places. The main difference noted in receiver instance model is the arc inscription that was applied. The inscription *MainSwitch.set_Power POn_MainSwitch On* assigns *On* to *Power* property for the variable *POn_MainSwitch* that represents the instance *MainSwitch*. This is achieved by applying the fourth step of the receiver instance.

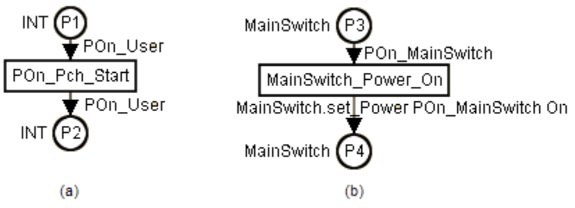


Figure 3.3: Representation of a minimum event

Among possible senders of a message, some possess a predictable behavior and others are unpredictable. The *User* and the *Environment* are actors that generate an unpredictable sequence of actions. These can execute enabled actions, actions that violate the scenario, as well as operations that were not specified in the current scenario. According to this, the steps presented below are needed to map this kind of message.

For the sender instance, which is the *User* or the *Environment*, it should:

1. create a transition to represent the enabled event, whose label should be formed by the name of the receiver instance, followed by the constant "_", accompanied by the name of the property that is being modified, followed by the constant "_", accompanied by the value that is being attributed to the property;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;

3. assign the CPN variable that identifies the instance to the arc inscriptions of the transition created in Step 1;

4. create a transition for each violating event, whose label should be formed by the name of the receiver instance, followed by the constant "_", accompanied by the name of the property that is being modified, followed by the constant "_NE" (*not enabled*). The input place of this transition is the same input place created in the second step;

5. create an output place for each transition created in the previous step, whose label should be composed by the name of the diagram, accompanied by the constant "_Stop" if the message is *cold* (the chart is stopped without a violation) or by the constant "_Abort" if the message is *hot* (the chart is aborted indicating a violation on the requirements);

6. assign the CPN variable that identifies the instance to the arc inscriptions of each transition created in Step 4.

**Definition 3.3.3.** *(Function for mapping an event for sender instance) Let* $LABP$ *be a function that assigns a label to a place,* $NS_i^1 \in N$ *be the CPN model for enabled event, and* $NS_i^2 \in N$ *be the CPN model for violating event if exists. A function* $MUE_S : l_L \mapsto N$*, that maps instance's localizations to a CPN structure of enabled and violating events for sending instance with the User or the Environment as sender, must be applied, where* $m_i.Src = User \wedge m_i.Src = Env$*, and* $evnt(loc([m_i, Send]))$ *is an enabled event* $e_e$ *or is a violating event* $e_v$*.*

$NS_i^1$ *is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Src}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Src})$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus \text{“\_”} \oplus m_i.m_S.P.Name \oplus \text{“\_”} \oplus m_i.m_S.V.$

*And $NS_i^2$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Src}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Src})$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ *"_"* $\oplus\ m_i.m_S.P.Name \oplus$ *"_NE"*;
- $LABP(p_{ii+1}) = \begin{cases} Nm(L) \oplus \text{"\_Stop"}, & \text{if } m_i \text{ is cold;} \\ Nm(L) \oplus \text{"\_Abort"}, & \text{if } m_i \text{ is hot.} \end{cases}$

For receiver instances, it should:

1. create a transition to represent the enabled event, whose label is formed by the name of the receiver instance, followed by "_", accompanied by the property's name that is being modified, followed by "_", accompanied by the value that is being assigned to the property;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the receiver instance;

3. assign the CPN variable that identifies the instance to the inscription of the input arc of the transition created in Step 1;

4. assign the expression *cs.set_idi c v* to the inscription of the output arc of the transition, where *cs* is the type that represents the instance, *idi* is the property's name that is being modified, *c* is the variable that represents the instance and *v* is the value that is being assigned to the property;

5. create a transition for each violating event, whose label should be formed by the name of the receiver instance, followed by the constant "_", accompanied by the name of the property that is being modified, followed by the constant "_NE" (*not enabled*). The input place of this transition is the same input place created in Step 2;

6. create an output place for each transition created in the previous step, whose label should be composed by the name of the diagram, accompanied by the constant "_Stop" if the message is *cold* (the chart is stopped without a violation) or by the constant "_Abort" if the message is *hot* (the chart is aborted indicating a violation on the requirements);

7. assign to the arc inscriptions of each transition created in Step 5, the CPN variable that identifies the instance.

**Definition 3.3.4.** *(Function for mapping an event for receiver instance) Let $NR_i^1 \in N$ be the CPN model for enabled event, $NR_i^2 \in N$ be the CPN model for violating event if exists. A function $MUE_R : l_L \mapsto N$, that maps instance's localizations to a CPN structure of enabled and violating events for receiving instance with the User or the Environment as sender, must be applied, where $m_i.Src = User \wedge m_i.Src = Env$, and $evnt(loc([m_i, Rcv])$ is an enabled event $e_e$ or is a violating event $e_v$.*

*$NR_i^1$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Dst}.O)$;
- $E((p_{ii}, t_{ii})) = M_O(m_i.i_{Dst})$;
- $E((t_i, p_{ii+1})) = M_{DTI}(m_i.i_{Dst}.O) \oplus$ "set_" $\oplus\ m_i.m_S.P.Name \oplus$ " " $\oplus\ M_O(m_i.i_{Dst}) \oplus$ " " $\oplus\ m_i.m_S.V$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ "_" $\oplus\ m_i.m_S.P.Name \oplus$ "_" $\oplus\ m_i.m_S.V$.

*And $NR_i^2$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Dst}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Dst})$;

- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ *"_"* $\oplus\ m_i.m_S.P.Name \oplus$ *"_NE";*
- $LABP(p_{ii+1}) = \begin{cases} Nm(L) \oplus \text{"\_Stop"}, & \text{if } m_i \text{ is cold;} \\ Nm(L) \oplus \text{"\_Abort"}, & \text{if } m_i \text{ is hot.} \end{cases}$

Figure 2.7 presents an LSC scenario, in which, according to the current *cut* (hatched line), the event *Switch2.Change(Med)* is enabled and the event *Switch1.Change(Med)* causes a violation if it occurs before *Switch2.Change(Med)*. Figure 3.4 shows, respectively, the CPN model of *User*, *Switch2*, *Switch1* for the message *Switch2.Change(Med)*. With this current *cut*, the message *Switch2.Change(Med)* is enabled (appears immediately after this cut) and the message *Switch1.Change(Med)* is a violating event (does not appear immediately after this cut), according to the definitions presented early. In Figure 3.4, firing the transition *Switch1_State_NE* indicates that the violating event occurs. The enabled event, in this case the message *Switch2.Change(Med)*, is represented by the transition *Switch2_State_Med*.

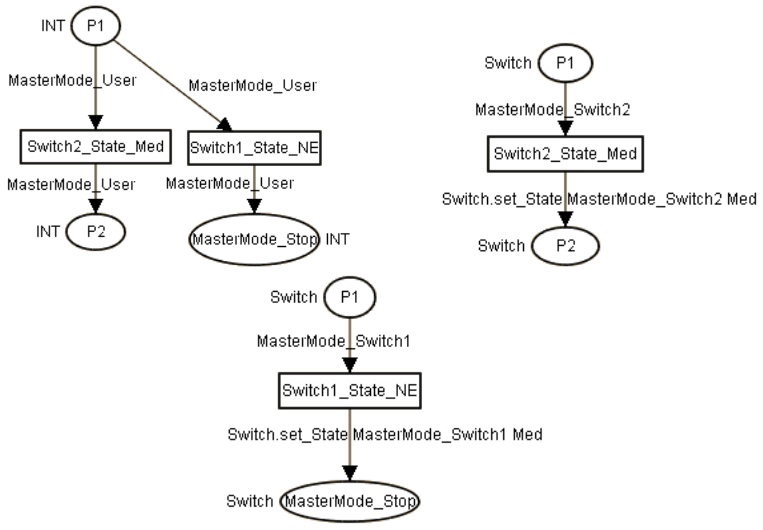


Figure 3.4: CPN model for the message *Switch2.Change(Med)*

Besides *User* and *Environment*, a message has system objects that could be a message sender. Those objects have a certain behavior that can be predictable, so the process to obtain the corresponding CPN model is simpler than *User* and *Environment* models. The steps that should be taken are the same as those applied when modeling the receiver instance of a message that represents a minimum event.

A LSC message represents a synchronous or an asynchronous communication. Both possess sending and receiving locations, however there is a difference in the execution order of these locations between these two types of messages. In the synchronous message, sending and receiving locations have the same execution order, i.e., the sending action and reception of the message happen at the same time. On the other hand, in the case of the asynchronous messages, the sending location has a precedence over the receiving location, indicating that the message is received after it is sent. In order to find a CPN model that represents an asynchronous message, the steps described next must be followed.

For the sender instance, it should:

1. create a transition, whose label is formed by the constant "SND_", accompanied by the name of the receiver instance, followed by the constant "_", accompanied by the name of the property that is being modified, followed by the constant "_", accompanied by the value that should be assigned to the property;
2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;
3. assign to the arc inscriptions of the transition created in Step 1, the variable that identifies the instance.

**Definition 3.3.5.** *(Function for mapping an asynchronous message for sender instance) Let $l_L$ be the set of locations of a chart $L$, $l_i \in l_L$ be a location of chart $L$, $M_L^A$ be the set of asynchronous message of a chart $L$ and $m_i \in M_L^A$ be an asynchronous message. A function $MAS_S : l_L \mapsto N$ that maps an instance's localization for sending event of an asynchronous message to a CPN structure must be applied, where $l_i \in loc([m_i, Send])$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Src}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Src})$;
- $LAB(t_{ii})$ = *"SND_"* $\oplus\ m_i.i_{Dst}.O.Name \oplus$ *"_"* $\oplus\ m_i.m_S.P.Name \oplus$ *"_"* $\oplus\ m_i.m_S.V$.

For the receiver instance, it should:

1. create a transition, whose label is formed by the constant "RCV_", accompanied by the name of the receiver instance, followed by "_", accompanied by the property's name that is being modified, followed by the constant "_", accompanied by the value that is being assigned to the property;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the receiver instance;

3. assign the CPN variable that identifies the instance to the inscription of the input arc of the transition;

4. assign the expression *cs.set_idi c v* to the inscription of the output arc of the transition, where *cs* is the type that represents the instance, *idi* is the property's name that is being modified, *c* is the variable that represents the instance and *v* is the value that is being assigned to the property.

**Definition 3.3.6.** *(Function for mapping an asynchronous message for receiver instance) Let* $l_L$ *be the set of locations of a chart* $L$*,* $l_i \in l_L$ *be a location of chart* $L$*,* $M_L^A$ *be the set of asynchronous message of a chart* $L$ *and* $m_i \in M_L^A$ *be an asynchronous message. A function* $MAS_R : l_L \mapsto N$ *that maps an instance's localization for receiving event of an asynchronous message to a CPN structure must be applied, where* $l_i \in loc([m_i, Recv])$*.* $N_i \in N$ *is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Dst}.O)$;
- $E((p_{ii}, t_{ii})) = M_O(m_i.i_{Dst})$;
- $E((t_{ii}, p_{ii+1})) = M_{DTI}(m_i.i_{Dst}.O) \oplus$ *"set_"* $\oplus\ m_i.m_S.P.Name \oplus$ " " $\oplus\ M_O(m_i.i_{Dst}) \oplus$ " " $\oplus\ m_i.m_S.V$;
- $LAB(t_{ii}) =$ *"RCV_"* $\oplus\ m_i.i_{Dst}.O.Name \oplus$ *"_"* $\oplus\ m_i.m_S.P.Name \oplus$ *"_"* $\oplus\ m_i.m_S.V$.

The CPN models for sender and receiver of an asynchronous message must be linked by a place, that acts like a *buffer*, whose label is formed by the constant "BUF_", accompanied by the name of the receiving instance, followed by the constant "_", accompanied by the name of the property that is being modified, followed by the constant "_", accompanied by the value that should be assigned to the property. This place must be of *int* type, where the input arc, that comes from the transition responsible for the sending event, has the constant value "1" in its inscription, to denote that a resource is being passed. The output arc of this place has the same inscription of the input arc.

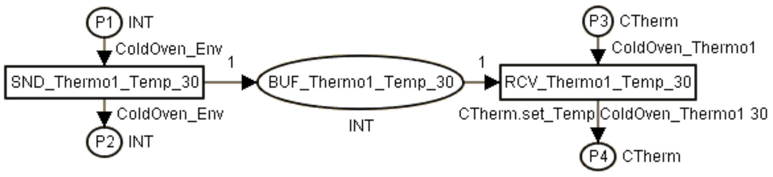


Figure 3.5: *CPN* model of an asynchronous message

Figure 3.5 shows the CPN model of the asynchronous message presented in Figure 2.6. One should observe that the *Environment* instance sends the message *SND_Thermo1_Temp_30* and continues its execution, and the *Thermo1* instance waits for a resource in the *buffer* and continues its execution after receiving the message *RCV_Thermo1_Temp_30*.

Usually, it is natural to specify more general scenarios. LSC language has symbolic messages to allow the modeling of such scenarios. When the scenario is in symbolic mode, the values shown in messages labels are replaced by variables. The first occurrence of a variable affects its value while the subsequent occurrences use the value that was assigned. There are two situations to concern on, when modeling this type of message. The steps to model a symbolic message are presented next.

For the sender instance, it should:

1. create a transition, whose label is formed by the name of the receiver instance, followed by "_", accompanied by the property's name that is being modified, followed by "_", accompanied by the LSC variable;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;

3. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition.

**Definition 3.3.7.** *(Function for mapping a symbolic message for sender instance) A function $MS_S : l_L \mapsto N$ that maps an instance's localization for the sending event of a symbolic message to a CPN structure must be applied, where $l_i \in loc([m_i, Send]) \wedge m_i.m_S.Symbolic = True$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Src}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Src})$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ "_" $\oplus\, m_i.m_S.P.Name \oplus$ "_" $\oplus\, m_i.m_S.V$.

For the receiver instance, it should:

1. create a CPN variable to represent the LSC variable, whose label is formed by the name of the receiver instance, followed by "_", accompanied by the LSC variable label;

2. create a transition, whose label is formed by the name of the receiver instance followed by "_", accompanied by the property's name that is being modified, followed by "_", accompanied by the LSC variable;

3. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the receiver instance;

4. assign the CPN variable that identifies the instance to the inscription of the input arc for the transition created on Step 2;

5. assign the expression *cs.set_idi c v* to the inscription of the output arc of the transition, where *cs* is the type that represents the instance, *idi* is the property's name that is being modified, *c* is the variable that represents the instance and *v* is the variable name;

6. for the first occurrence of a variable:

- create one output place for the transition created in Step 2. The type of this place should be the type created to represent the LSC variable, whose label is the variable name created in Step 1;
- assign the variable created in the first step to the inscription of the output arc of the place created in previous step.

7. for subsequent occurrences of a variable:

   - create a place that is an output and an input for the transition created in Step 2. The type of this place should be the type created to represent the LSC variable, whose label is the variable name created in Step 1;
   - assign the variable created in Step 1 to the inscription of the output arc and input arc of the place created in previous step.

**Definition 3.3.8.** *(Function for mapping a symbolic message for receiver instance) Let* $v_i \in V_L$ *be an LSC variable. A function* $MS_R : l_L \mapsto N$ *that maps an instance's localization for the receiving event of a symbolic message to a CPN structure must be applied, where* $l_i \in loc([m_i, Rcv]) \wedge m_i.m_S.Symbolic = True$. $N_i \in N$ *is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}, p_{ii+2}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (t_{ii}, p_{ii+2})\}$, *if it is the first occurrence of the variable, otherwise* $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (t_{ii}, p_{ii+2}), (p_{ii+2}, t_{ii})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Dst}.O)$;
- $C(p_{ii+2}) = M_{DT}(m_i.m_S.V)$;
- $E((p_{ii}, t_{ii})) = M_O(m_i.i_{Dst})$;
- $E((t_{ii}, p_{ii+1})) = M_{DTI}(m_i.i_{Dst}.O) \oplus$ *"set_"* $\oplus\ m_i.m_S.P.Name \oplus$ " " $\oplus\ M_O(m_i.i_{Dst}) \oplus$ " " $\oplus\ M_O(m_i.i_{Dst})$;
- $E((t_{ii}, p_{ii+2})) = MV(v_i)$;
- $E((p_{ii+2}, t_{ii})) = MV(v_i)$, *if it is not the first occurrence of the variable;*
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ "_" $\oplus\ m_i.m_S.P.Name \oplus$ "_" $\oplus\ m_i.m_S.V$;
- $LABP(p_{ii+2}) = Nm(L) \oplus$ "_" $\oplus\ m_i.m_S.V$.

Take a look at the example of symbolic messages presented in Figure 3.6. The goal of this scenario is to assign the same state that was applied to the instance *MainSwitch* for the instance *MainLight*, i.e., when the user activates *MainSwitch*, then *MainLight* is also activated. When the user disables *MainSwitch*, then *MainLight* is also disabled. In this scenario, it is possible to represent two situations in just one LSC diagram. Figure 3.7 depicts the CPN model obtained for the first occurrence of the variable *Xs* in the message *Click(Xs)* by applying the aforementioned steps.

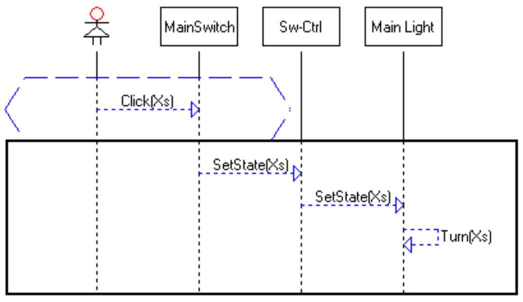


Figure 3.6: LSC chart with symbolic messages

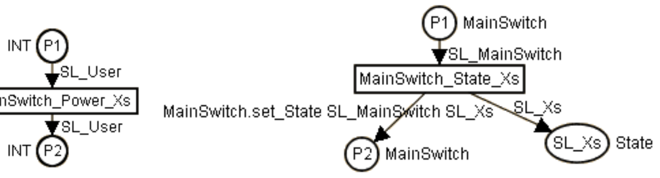


Figure 3.7: CPN model for the first occurrence of the variable *Xs* for sender and receiver instances

The messages seen up to now modify the property value of an object, but other message type allows to transfer data or control signs between objects. Such message is related with method calls. Figure 3.6 presents a scenario with two method calls. *MainSwitch* invokes the method *SetState(Xs)* of *Sw-Crtl*, that passes the information ahead calling the method *SetState(Xs)* of *MainLight*.

In order to obtain the CPN model for method calls, the steps described next must be followed.

For the sender instance, it should:

1. create a transition, whose label is formed by the name of the receiver instance, followed by "_", accompanied by the method's name;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;
3. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition.

**Definition 3.3.9.** *(Function for mapping a method call for sender instance) Let $l_i \in l_L$ be a location of chart $L$, $m_i \in M_L$ be an LSC message of a chart $L$, $m_i.m_S.m$ be a method and $m_i.m_S.m.Name$ be the method's name. The function $MFMC_S : l_L \mapsto N$ maps an instance's localization for sending event of a method call to a CPN structure, where $l_i \in loc([m_i, Send]) \wedge m_i.m_S.m \neq \perp$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Src}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Src})$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ "_" $\oplus\ m_i.m_S.m.Name$.

For the receiver instance, it should:

1. create a transition, whose label is formed by the name of the receiver instance, followed by "_", accompanied by the method's name;
2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;
3. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition;
4. create one input place to the transition created in Step 1 for representing each parameter used in the method call. The type of these places should be the type of the parameter;
5. in the case of an exact message (message that uses a constant value instead of a variable), assign the constant value to the inscriptions of the input and the output arcs of the places created in the previous step. Otherwise applies the steps applied to a symbolic message.

**Definition 3.3.10.** *(Function for mapping a method call for receiver instance) Let $l_i \in l_L$ be a location of chart $L$, $m_i \in M_L$ be an LSC message of a chart $L$ and $v_i \in V_M$ be the parameter of method $m_i.m_S.m$. The function $MFMC_R : l_L \mapsto N$ maps an instance's localization for receiving event of a method call to a CPN structure, where $l_i \in loc([m_i, Recv]) \wedge m_i.m_S.m \neq \perp$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}, p_{ii+2}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (p_{ii+2}, t_{ii})\}$, *if* $m_i.m_S.Symbolic = False$ *or* $\{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (p_{ii+2}, t_{ii}), (t_{ii}, p_{ii+2})\}$, *if* $m_i.m_S.Symbolic = True$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Dst}.O)$;
- $C(p_{ii+2}) = M_{DT}(v_i)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Dst})$;
- $E((t_{ii}, p_{ii+2})) = MV(v_i)$, *if* $m_i.m_S.Symbolic = True$;
- $E((p_{ii+2}, t_{ii})) = MV(v_i)$, *if* $m_i.m_S.Symbolic = True$ *or* $v_i$, *if* $m_i.m_S.Symbolic = False$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ "_" $\oplus$ $m_i.m_S.m.Name$.

Figure 3.8 shows the CPN models of *SwCtrl.SetState(Xs)* method call, for sender and receiver instances (see Figure 3.6) obtained by applying the steps described earlier.

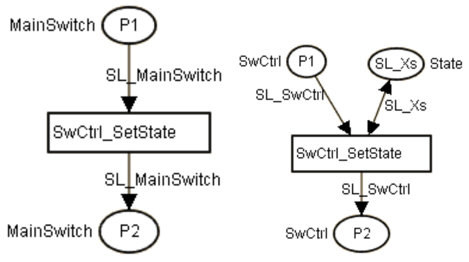


Figure 3.8: CPN model of *SwCtrl.SetState(Xs)* method call for sender and receiver instances

## 3.4 External Functions

The *Play-Engine* tool [24] allows applications with graphical interface supply external functions.

In order to obtain a CPN model that represents an external function call, the next steps must be followed.

If sender and receiver are different instances, it should:

1. create a transition, whose label is formed by the chart's name, followed by "_", accompanied by the function's name;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;

3. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition.

**Definition 3.4.1.** *(Function for mapping a external function call for sender instance) Let $l_i \in l_L$ be a location of chart $L$, $m_i \in M_L$ be an LSC message of a chart $L$, $v_i \in V_M$ be a parameter of external function $m_i.m_S.f$ and $m_i.m_S.f.Name$ the external function's name. The function $MFEF_S : l_L \mapsto N$ maps an instance's localization for sending event of an external function call to a CPN structure, where $l_i \in loc([m_i, Send]) \wedge m_i.m_S.f \neq \perp$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Src}.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Src})$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ "_" $\oplus\ m_i.m_S.f.Name$.

The steps below must be executed for the receiver instance or when the sender and receiver are the same instance:

1. create a transition, whose label is formed by the chart's name, followed by "_", accompanied by the function's name;

2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the sender instance;

3. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition;

4. if the function has a variable as parameter, create a CPN variable, whose name is composed of the chart's name, followed by "_", accompanied by the name of the LSC variable;

5. create a place to represent the function parameter. If this parameter is a variable, then the label of the place must be composed of the chart's name, followed by the constant "_", accompanied by the name of the variable created in the previous step. These places are input and output to the transition created in Step 1;

6. if the function has a variable as parameter, assign the corresponding variable created in Step 4 to the inscriptions of the arcs that arrive and leave the places created in the previous step. If the function has constants values as parameters, these values are assigned to the inscriptions of these arcs;

7. create a variable to represent the result of the function. The name of this variable must be formed by "RST_", followed by the chart's name, accompanied by "_", followed by the function's name. The type of this variable must be equivalent to the result type of the function;

8. create one output place for the transition created in Step 1, whose label must be formed by "RST_", followed by the chart's name, accompanied by "_", followed by the function's name. The type of this place must be equivalent to the result type of the function;

9. assign the variable created in Step 7 to the inscription of the arc that arrive in the place created in the previous step.

**Definition 3.4.2.** *(Function for mapping a external function call for receiver instance) Let $l_i \in l_L$ be a location of chart $L$, $m_i \in M_L$ be an LSC message of the chart $L$, $v_i \in V_M$ be a parameter of external function $m_i.m_S.f$. The function $MFEF_R : l_L \mapsto N$ maps an instance's localization for receiving event of of an external function call to a CPN structure, where $l_i \in loc([m_i, Recv]) \wedge m_i.m_S.f \neq \perp$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;

- $P = \{p_{ii}, p_{ii+1}, p_{ii+2}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (p_{ii+2}, t_{ii})\}$, *if* $m_i.m_S.Symbolic = False$ *or* $\{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (p_{ii+2}, t_{ii}), (t_{ii}, p_{ii+2})\}$, *if* $m_i.m_S.Symbolic = True$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(m_i.i_{Dst}.O)$;
- $C(p_{ii+2}) = M_{DT}(v_i)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(m_i.i_{Dst})$;
- $E((t_{ii}, p_{ii+2})) = MV(v_i)$, *if* $m_i.m_S.Symbolic = True$;
- $E((p_{ii+2}, t_{ii})) = MV(v_i)$, *if* $m_i.m_S.Symbolic = True$ *or* $v_i$, *if* $m_i.m_S.Symbolic = False$;
- $LAB(t_{ii}) = m_i.i_{Dst}.O.Name \oplus$ *"_"* $\oplus\ m_i.m_S.f.Name$;
- $LABP(p_{ii+2}) =$ *"RST_"* $\oplus\ Nm(L) \oplus$ *"_"* $\oplus\ m_i.m_S.f.Name$.

Figure 3.9 shows the CPN model that represents the external function presented in Figure 2.8. Places *ShowSum_X174* and *ShowSum_X176* represent the variables *X174* and *X176*, respectively. One should observe that these places are input and output for the transition *ShowSum_Sum*. Therefore they represent the second occurrence of the respective variables. A *token* in place *RST_ShowSum_Sum*, after firing the transition *ShowSum_Sum*, represents the returned value of the external function.

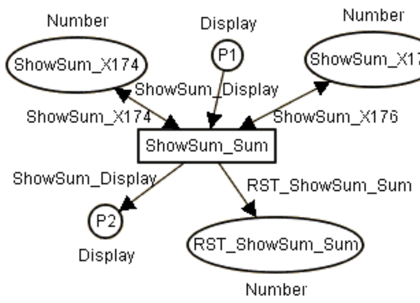


Figure 3.9: CPN model that represents an external function

## 3.5 Assignments

An assignment is an LSC construction that allows storing properties' values, constant values or a result of a external function, for a subsequent use inside the chart.

In order to obtain the CPN model for an assignment that allows storing properties' values, the steps below should be executed for all of the instances that are synchronizing their activities with the assignment:

1. create a CPN variable of the same type of the property that is being stored, whose name is composed of the chart's name, followed by "_", accompanied by LSC variable;
2. create a transition, whose label is formed by the chart's name, followed by "_AS_ID", where *ID* is a integer sequential number, larger than zero;
3. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the instance;
4. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition;
5. for the instance whose property is been saved, create an output place for the transition created in Step 2, whose type is the type of the property that is being saved. The label of this place is composed by the chart's name, followed by "_", accompanied by the name of the variable that is saving the value;
6. assign *#id VarName* to the inscription of the arc that arrives in the place created in the previous step, where *id* is the property that been saved and *VarName* is the variable that represents the instance.

**Definition 3.5.1.** *(Function for mapping an assignment that stores a property value) Let $p_x^i$ be a property of an instance $i_i \in I_A$, which is synchronized with an assignment $a_i$ and $a_i.P$ be the property that should be stored. A function $MASP : I_A \mapsto N$ must be applied to instances that synchronizes with an assignment that stores a property value in order to obtain a CPN model, where $\exists p_x^i : p_x^i = a_i.P$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}, p_{ii+2}\}$;

- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (t_{ii}, p_{ii+2})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$;
- $C(p_{ii+2}) = M_{DT}(p^i_x.D)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(i_i)$;
- $E((t_{ii}, p_{ii+2})) = \#a_i.P.Name\ M_{DT}(a_i.P.D)$;
- $LABP(p_{ii+2}) = Nm(L) \oplus$ *"_"* $\oplus\ a_i.V$;
- $LAB(t_{ii}) = Nm(L) \oplus$ *"_AS_ID",*

  *where $ID \in \mathbb{N}^+$, $p^i_x.D$ represents the domain of a property of a synchronized instance, $a_i.P$ is the stored property, $a_i.P.Name$ is the property's name, $a_i.P.D$ represents the domain of the stored property, and $a_i.V$ is the variable which contains the assigned value.*

Figure 3.10 depicts the CPN model obtained for the assignment *N1 := Display.Value* presented in Figure 2.9. The first model on the left hand side represents instance *Display* and the second one depicts the model of instance *Plus*. One should observe the difference between these models. In the first model (*Display*), there is a place *SS_N1* of the type *STRING* that represents the variable *N1*, which contains the property value of *Display* instance. In the second model (*Plus*) there is no such place, because the instance *Plus* is just synchronizing its activities with the assignment.

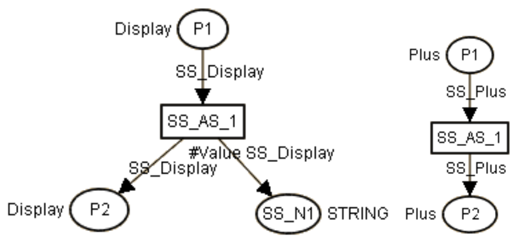


Figure 3.10: CPN model of an assignment that stores an object property value

When the expression on the right side of the assignment operator is a function result, the place that represents the function result (Section 3.4) must be an input place of the transition created to represent the assignment construction, so the following steps should be taken:

1. create a CPN variable of the same type of the function result, whose name is composed by the chart's name, followed by "_", accompanied by LSC variable;

2. create a transition, whose label is formed by the chart's name, followed by "_AS_ID", where *ID* is a integer sequential number, larger than zero;

3. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the instance;

4. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition;

5. the place that represents the function result must be an input place of the transition created in Step 2;

6. create an output place for the transition created in Step 2, whose type is the type of the function result. The label of this place should be composed by the chart's name, followed by "_", accompanied by the name of the variable that is saving the value;

7. assign the variable created in Step 1 to the inscription of the arc that arrives in the place created in the previous step.

**Definition 3.5.2.** *(Function for mapping an assignment that stores a function result) Let $a_i \in A_L$ be an assignment of chart $L$, $i_i \in I_A$ be an instance that is synchronized with the assignment $a_i$, $a_i.f$ is an external function used in the assignment $a_i$ and $a_i.f.d_f$ is the result value of the external function $a_i.f$. The function $MAFR : I_A \mapsto N$ maps each of the instances that synchronizes with an assignment that stores a function result to a CPN model, where $a_i.f \neq \bot$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}, p_{ii+2}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (t_{ii}, p_{ii+2})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$;
- $C(p_{ii+2}) = M_{DT}(a_i.f.d_f)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(i_i)$;

- $E((t_{ii}, p_{ii+2})) = MV(a_i.V)$;
- $LABP(p_{ii+2}) = Nm(L) \oplus$ *"_"* $\oplus\ a_i.V$;
- $LAB(t_{ii}) = Nm(L) \oplus$ *"_AS_ID"*.

And if the expression on the right side of the assignment is a constant, then it should:

1. create a transition, whose label is formed by the chart's name, followed by "_AS_ID", where *ID* is an unique sequential number;
2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the instance;
3. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition;
4. create an output place for the transition created in Step 1, whose type is the type of the value that is being saved. The label of this place should be composed by the chart's name, followed by "_", accompanied by the name of the variable that is saving the value;
5. assign the constant value, presented on the right side of the assignment, to the inscription of the arc that arrives in the place created in the previous step.

**Definition 3.5.3.** *(Function for mapping an assignment that stores a constant value) Let $a_i \in A_L$ be an assignment of chart $L$, $i_i \in I_A$ be an instance that is synchronized with the assignment $a_i$ and $a_i.C$ is the constant used in the assignment $a_i$. The function $MACV : I_A \mapsto N$ maps each instance that synchronizes with an assignment that stores a constant value to a CPN model, where $a_i.C \neq \bot$. $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}, p_{ii+2}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1}), (t_{ii}, p_{ii+2})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$;
- $C(p_{ii+2}) = M_{DT}(a_i.C)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(i_i)$;

- $E((t_{ii}, p_{ii+2})) = a_i.C$;
- $LABP(p_{ii+2}) = Nm(L) \oplus$ "_" $\oplus a_i.V$;
- $LAB(t_{ii}) = Nm(L) \oplus$ "_AS_ID".

## 3.6 Conditions

A condition represents a decision structure that can be composed of a conjunction of expressions and can be evaluated as true or false.

In order to obtain the CPN model for each instance that synchronizes with a condition, the following steps must be taken:

1. create a transition, whose label is composed by the chart's name, followed by "_CD_ID_SYNC", where *ID* is an unique sequential number;
2. create one input and one output places for the transition created in the previous step. The type of these places should be the type created to represent the instance;
3. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transition.

**Definition 3.6.1.** *(Function for mapping synchronization points at a condition) Let $i_i \in I_C$ be an instance that synchronizes with the condition $c_i$ and $i_i.O$ be the concrete object of the synchronized instance $i_i$. The function $MCSP : I_C \mapsto N$ must be applied to each instance that synchronizes with a condition in order to obtain a CPN model, where $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}\}$;
- $P = \{p_{ii}, p_{ii+1}\}$;
- $A = \{(p_{ii}, t_{ii}), (t_{ii}, p_{ii+1})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = M_O(i_i)$;
- $LAB(t_{ii}) = Nm(L) \oplus$ "_CD_ID_SYNC"
  *, where $ID \in \mathbb{N}^+$.*

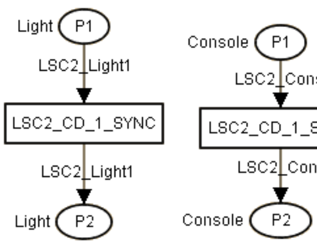


Figure 3.11: CPN model for synchronization point of a condition

Figure 3.11 shows the CPN models of the instances *Light1* and *Console*, for the condition *Light1=Green* presented in Figure 2.10, respectively. As condition *Light1=Green* has two instances synchronizing their activities, then it is necessary to model the synchronization point.

In order to obtain the CPN model of an instance which has a property constrained by a condition using some comparison operator with a constant value, another variable or a function result, the following steps must be taken:

1. create a transition to represent the true value, whose label is composed by the chart's name, followed by "_CD_ID_TRUE", where *ID* is an unique sequential number;

2. assign the guard condition "*#id VarName oper value*" to the transition created on the previous step, where *id* is the property that is being compared, *VarName* is the variable that represents the instance, *oper* is the relational operator used in the condition expression, and *value* is the value that is being compared, which can be a constant value, another variable or a function result;

3. create a transition to represent the false value, whose label is composed by the chart's name, followed by "_CD_ID_FALSE", where *ID* is the same used in Step 1;

4. assign the guard condition "*#id VarName oper value*" to the transition created on the previous step, where *id* is the property that is being compared, *VarName* is the variable that represents the instance, *oper* is the opposite relational operator used in the condition expression, and *value* is the value that is being compared, which can be a constant value, another variable or a function result;

5. create a common input place for the transitions created in the previous steps. The type of this place should be the type created to represent the instance;

6. create an output place for the transition created in Step 1. The type of this place should be the type created to represent the instance;

7. create an output place for the transition created in Step 3. The type of this place should be the type created to represent the instance. This place is not the same as the input place for this transition if it is a *cold* condition, or if it is a FALSE condition, or still if it is a hot condition used at the end of *Vertical Delay*, *Message Delay* and *Timer* time restrictions, presented later. This step is necessary in order to guarantee that a *hot* condition should be tested until evaluates to true, so it can be executed. The cases defined above are exceptional situations of a *hot* condition that produce a false value when the condition will never be able to be evaluated to true;

8. assign the CPN variable that identifies the instance to the inscriptions of the input and the output arcs of the transitions created in Steps 1 and 3.

**Definition 3.6.2.** *(Function for mapping a cold condition) Let $\varphi_i \in c_i.\varphi$ be a basic expression of an instance $i_i \in I_C$, which is synchronized with the condition $c_i$. The function $MCC_{EXP} : c_i.\varphi \mapsto N$ must be applied to an expression of a cold condition in order to obtain a CPN model, where $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}, t_{ii+1}\}$;
- $P = \{p_{ii}, p_{ii+1}, p_{ii+2}\}$;
- $A = \{(p_{ii}, t_{ii}), (p_{ii}, t_{ii+1}), (t_{ii}, p_{ii+1}), (t_{ii+1}, p_{ii+2})\}$;
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$;
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = E((t_{ii+1}, p_{ii+2})) = M_O(i_i)$;
- $G(t_{ii})$ =*"#$i_i.O.p_i$ $M_{DTI}(i_i.O)$ $Oper(\varphi_i)$ $RHS(\varphi_i)$"*;
- $G(t_{ii+1})$ =*"#$i_i.O.p_i$ $M_{DTI}(i_i.O)$ $NOper(\varphi_i)$ $RHS(\varphi_i)$"*;
- $LAB(t_{ii}) = Nm(L) \oplus$ *"_CD_ID_TRUE"*;

- $LAB(t_{ii+1}) = Nm(L) \oplus$ *"_CD_ID_FALSE",*

  *where $i_i.O.p_i$ is a property of the constrained instance, $i_i.O$ represents the concrete object of the constrained instance, $ID \in \mathbb{N}^+$, $Oper$ is a function that returns the relational operator of a basic expression, $NOper$ is a function that returns the opposite relational operator used in the basic expression, and $RHS$ is a function that returns the binding expression of a basic expression.*

**Definition 3.6.3.** *(Function for mapping a hot condition) A function $MCH_{EXP} : c_i.\varphi \mapsto N$ must be applied to an expression of a hot condition, in order to obtain a CPN model, where $N_i \in N$ is defined as follows:*

- $T = \{t_{ii}, t_{ii+1}\}$*;*
- $P = \{p_{ii}, p_{ii+1}\}$*;*
- $A = \{(p_{ii}, t_{ii}), (p_{ii}, t_{ii+1}), (t_{ii}, p_{ii+1}), (t_{ii+1}, p_{ii})\}$*;*
- $C(p_{ii}) = C(p_{ii+1}) = M_{DTI}(i_i.O)$*;*
- $E((p_{ii}, t_{ii})) = E((t_{ii}, p_{ii+1})) = E((t_{ii+1}, p_{ii})) = M_O(i_i)$*;*
- $G(t_{ii}) =$*"$\#i_i.O.p_i\ M_{DTI}(i_i.O)\ Oper(\varphi_i)\ RHS(\varphi_i)$";*
- $G(t_{ii+1}) =$*"$\#i_i.O.p_i\ M_{DTI}(i_i.O)\ NOper(\varphi_i)\ RHS(\varphi_i)$";*
- $LAB(t_{ii}) = Nm(L) \oplus$ *"_CD_ID_TRUE";*
- $LAB(t_{ii+1}) = Nm(L) \oplus$ *"_CD_ID_FALSE",*
  *where $ID \in \mathbb{N}^+$.*

Figure 3.12 shows the CPN model that represents the expression *Light1=Green* of the *cold* condition presented in Figure 2.10. First, the steps to represent the synchronization point are applied. The transition *LSC1_CD_1_SYNC* represents this synchronization point. Places *P1* and *P3* are, respectively, input and output for *Light1* instance of *WarnLight* type, and places *P2* and *P4* are, respectively, input and output for *Console* instance. After mapping synchronization points, condition expressions must be modeled. Each transition has a guard condition that controls which transition should fire. One should observe that the guard conditions deny each other, so just one transition can fire.

Other condition expressions can be formed by the reserved words *TRUE, SYNC* or *FALSE*. These expressions have a defined value, so only synchronization points need to be modeled, as the condition does not need to be

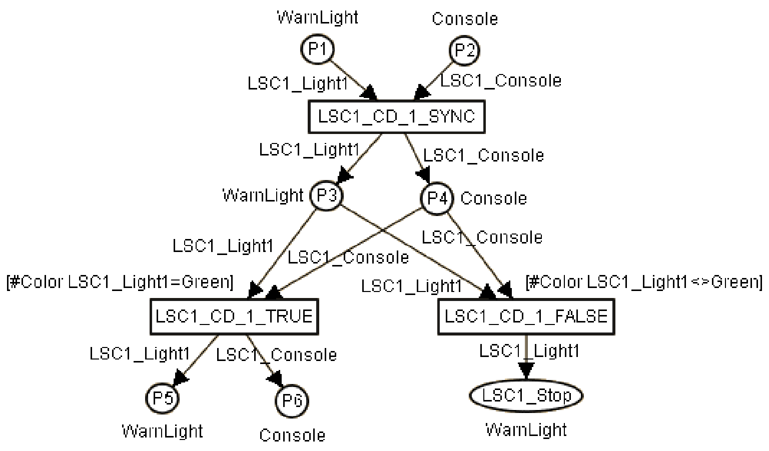


Figure 3.12: CPN model that represents a *cold* condition that constrains an object's property

evaluated. Therefore the steps that should be taken to obtain the CPN model are the same used to map the synchronization point of a condition.

Conditions expressions can also use variables. Those expressions accomplish comparison between variables or still between variable and a value, which can be a constant value or an evaluation result of a function call. As it was seen, functions are always modeled before the underlying element in which they are used (i.e. inside a message call, inside an assignment expression or as part of a condition expression). The steps adopted to obtain the CPN model that represents these types of expressions are similar to the steps considered to obtain the model that represents an expression that constrains a property's value of a certain object, in which the variable usage should be mapped according to the steps presented in Section 3.3, where symbolic messages are presented.

Figure 3.13 shows an LSC scenario with the condition $T < 30$, which uses the variable *T*, and the corresponding CPN model. The place *ColdOven_T* represents the variable *T* in the expression $T < 30$. One should observe that guard conditions reflect the constrains on *ColdOven_T* variable.

## 3.7 *If-then-else* Construction

The *if-then-else* construction allows different scenarios to be executed depending on a condition.

An *if-then-else* construction is formed by basic constructions (messages,

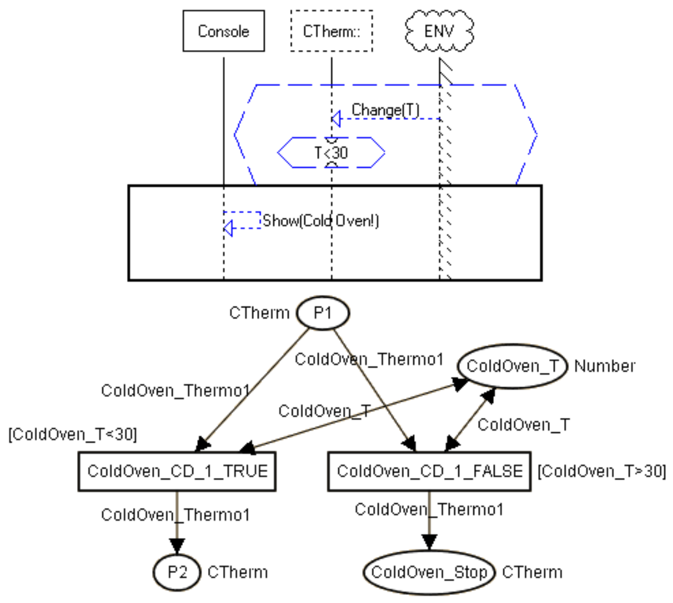


Figure 3.13: Condition using a variable and its corresponding CPN model

assignments, conditions, subcharts) inside specific scenarios (*then* or *else*), i.e., it does not involve any new construction, therefore the steps below should be followed in order to obtain the equivalent CPN model:

1. model the synchronization point of the beginning of *then* subchart ($loc([Sub_T, Start])$), according to rules presented in Section 3.2;

2. model the controlling condition, according to rules presented in Section 3.6;

3. model the construction of *then* scenario, according to the rules presented for each construction. The last construction is the synchronization point of the ending of *then* subchart ($loc([Sub_T, End])$);

4. if there is an *else* scenario, the synchronization point of the beginning of this subchart ($loc([Sub_E, Start])$) must be modeled first. After that, the constructions inside the scenario must be modeled, and finally, the synchronization point of the ending of *else* subchart ($loc([Sub_E, End])$);

5. the output places corresponding to the ending of each scenario ($loc([Sub_T, End]$) and $loc([Sub_E, End])$) should be united.

**Definition 3.7.1.** *(Function for mapping an if-then-else construction) Let $ITE$ be an if-then-else construction of chart $L$, $i_i, i_{i+1}, ..., i_m \in I_{ITE}$ be instances that participate in the scenario of $ITE$ construction, $l^i_j, l^i_{j+1}, ..., l^i_n$ be locations of instance $i_i$, $N_j, N_{j+1}, ..., N_n$ be the corresponding CPN models obtained for the locations $l^i_j, l^i_{j+1}, ..., l^i_n$ of instance $i_i$, $loc([Sub_T, Start])$, $loc([Sub_T, End])$, $loc([Sub_E, Start])$ and $loc([Sub_E, End])$ be the locations that represent, respectively, the beginning of then subchart, the ending of then subchart, the beginning of else subchart and the ending of else subchart. A function $MITE$ that maps an if-then-else construction, should:*

- *apply the mapping rules for each location $l^i_j, l^i_{j+1}, ..., l^i_n$, where $l^i_j$ is the first location to be mapped and $l^i_n$ is the last, according to the execution order described in Section 3.3;*
- *apply the joining process described in Section 3.11;*
- *consider two CPN models $N_a$ and $N_b$, where $N_a$ is the obtained model for the location $loc([Sub_T, End])$ and $N_b$ is the obtained model for the location $loc([Sub_E, End])$. Let $p_x$ be an output place for the transition created in the CPN model $N_a$ and $p_y$ be an output place for the transition created in the CPN model $N_b$, so rename $p_y$ to $p_x$ by applying the corresponding steps described in Section 3.11.*

By applying the above steps, Figure 3.14 depicts the CPN model of the *if-then-else* construction presented in Figure 2.11. The transition *TP_Sub_1_Start* represents the synchronization point of the *then* subchart, which must be modeled first. After that, the condition *PrbCtrl.Probing=True* is modeled, and it decides which scenario should execute, the *then* subchart or the *else* one. After modeling all constructions inside each scenario, transitions *TP_Sub_1_End* and *TP_Sub_2_End* are created to represent the synchronization point of the ending of *then* and *else* scenarios, respectively.

## 3.8 Loops

A *loop* construction allows the execution of a scenario several times.

A loop's scenario may have several constructions, so in order to find a CPN model that represents a loop construction, the mapping rules must be followed for each construction inside the loop's scenario, according to the execution order. After mapping all constructions, the last obtained place of the CPN model must be linked with the first obtained place of this CPN model in order to represent a loop iteration, as presented next.

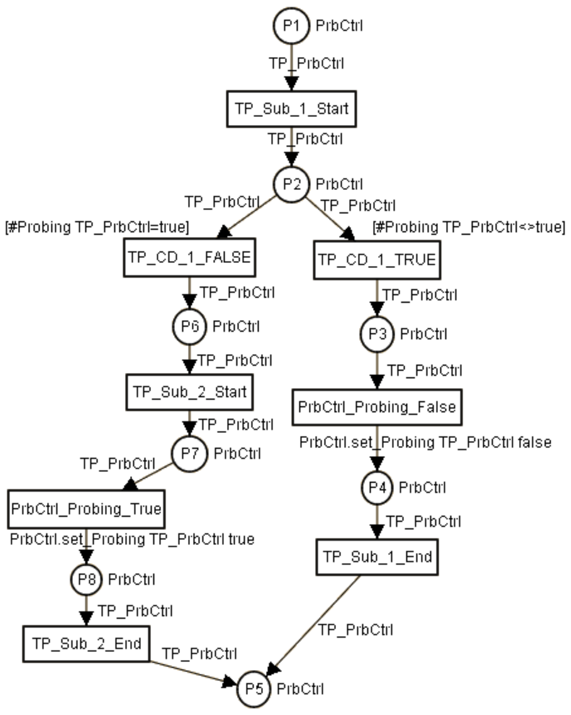


Figure 3.14: CPN model of an *if-then-else* construction

In order to obtain the CPN model for a *fixed* loop, the following steps should be taken:

1. create an *INT* variable to represent the loop's index. The name of this variable should be composed by the chart's name, followed by "_IND_ID", where *ID* uniquely identifies a subchart;

2. map the synchronization point of the beginning of the subchart ($loc([Sub_{LOOP}, Start])$, as it was presented in the Section 3.2. The transition created in this step must have the following segment code:

   - *output(variable) action(0)*, where *variable* is the variable created in the previous step;

3. create an *INT* place, whose label is composed by the variable's name, created in Step 1;

4. assign the variable created in Step 1 to the arc inscription, which comes from the transition created in the Step 2 to the place created in the previous step;

5. model the LSC constructions presented inside loop's scenario according to their mapping rules presented earlier;

6. assign the following guard condition to transition which represents the entry point ($loc([Sub_{LOOP}, Start])$ for each participating instance;

    - *[variable < max]*, where *variable* is the variable created in Step 1 and *max* is the number of iterations.

7. assign the following guard condition to the transition which represents the synchronization point at the ending of the loop's scenario ($loc([Sub_{LOOP}, End])$:

    - *[variable = max]*, where *variable* is the variable created in Step 1 and *max* is the number of iterations.

8. the transition created in the previous step is an input of the place created in Step 3 and the place generated when mapping the synchronization point at the beginning of the loop's scenario ($loc([Sub_{LOOP}, Start])$;

9. after modeling all constructions, create a transition, which returns to the beginning of the loop's scenario (a new iteration). The label of this transition is formed by the chart's name, followed by "_LOOP_ID", where *ID* is equals to the subchart *ID*.

**Definition 3.8.1.** *(Function for mapping a loop construction) Let $Loop$ be a loop construction of chart L, $i_i, i_{i+1}, ..., i_m \in I_{Loop}$ be instances that participate in the scenario of $Loop$ construction, $l^i_j, l^i_{j+1}, ..., l^i_n$ be locations of instance $i_i$, $N_j, N_{j+1}, ..., N_n$ be the corresponding CPN models obtained for the locations $l^i_j, l^i_{j+1}, ..., l^i_n$ of instance $i_i$, $v_i$ be a CPN variable created to represent the loop's index, $loc([Sub_{Loop}, Start])$ and $loc([Sub_{Loop}, End])$ be the locations that represent, respectively, the beginning and ending of loop subchart. A function $MLOOP$ that maps a loop construction, should:*

- *$\forall i_i \in I_{Loop}$ apply the mapping rules for each location $l^i_j, l^i_{j+1}, ..., l^i_n$, where $l^i_j$ is the first location to be mapped and $l^i_n$ is the last, according to the execution order described in Section 3.3;*

- *$LABV(v_i) = Nm_(L) \oplus$ "_IND_ID";*

- *let $p_z$ be a place created to represent the index variable, so $LABP(p_z) = Nm(L)$;*

- *let $N_a$ be the CPN model obtained for the location $loc([Sub_{LOOP}, Start])$ and $t_x$ be the transition created when mapping this location, so $SC(t_x) = output(v_i)\ action(0)$ and $N_a.E(t_x, p_z) = v_i$, where $SC$ is a function that assigns a segment code to a transition and $N_a.E$ is the arc function;*

- *let $N_b$ be the CPN model obtained for the location $loc([Sub_{LOOP}, Start])$ and $t_f$ be the transition created when mapping this location, so $N_b.G(t_f)$ =[$v_i < idx$], where $N_b.G$ is the function that assigns a guard to the transition and $idx \in \mathbb{N}^+$;*

- *let $N_c$ be the CPN model obtained for the location $loc([Sub_{LOOP}, End])$ and $t_g$ be the transition created when mapping this location, so $N_c.G(t_g)$ =[$v_i = idx$] and $N_c.A = N_c.A \cup \{(t_g, p_y), (t_g, p_z)\}$, where $N_c.G$ is the function that assigns a guard to the transition, $N_c.A$ is the set of arcs, $p_y \in N_a.P$ (set of places of $N_a$) be the input place for the transition created while mapping the location $loc([Sub_{LOOP}, Start])$ and $idx \in \mathbb{N}^+$;*

- *let $N_d$ be a CPN model created to represent a new loop iteration and $t_z$ be a transition of this model. Apply $LABT(t_z) = Nm(L) \oplus$ "_LOOP_ID", where $ID$ is the ID for $Sub_{LOOP}$ subchart. $N_d.A = \{(p_m, t_z), (t_z, p_y)\}$, where $p_m \in N_c.P$ (set of places of $N_c$) be an output place of the transition created when mapping the location $loc([Sub_{LOOP}, End])$ and $N_d.A$ be the set of arcs of $N_d$;*

- *apply the joining process described in Section 3.11,*

  *where $ID \in \mathbb{N}^+$ and $LABV$ is a function that assigns a label for a CPN variable.*

As a loop construction is formed by basic constructions, the formalism behind this construction is not presented. It can be easily mapped by applying the steps for each construction inside the scenario of the construction, following the execution order presented earlier.

By applying the above steps, Figure 3.15 shows the CPN model obtained for the *fixed* loop presented in Figure 2.13(a). When the transition *LSC1_Sub_1_Start* fires, the variable *LSC1_IND_1*(represents the index of the loop) is initialized and a *token* with the initial value is deposited in place *LSC1_IND_1*. This value is considered in the guard conditions of transitions *Light1_Color_Red* and *LSC1_Sub_1_End*, in order to decide if the loop should be iterated. This value is increased by firing transition *LSC1_LOOP_1*. This transition represents a loop iteration, where a *token* is deposited in place *P2*, which is an input of the first event of the scenario, the message *Change(Red)*. The loop scenario is executed until variable *LSC1_IND_1* reaches the value 3, where transition *LSC1_Sub_1_End* fires, leading the execution to the next location after the loop.

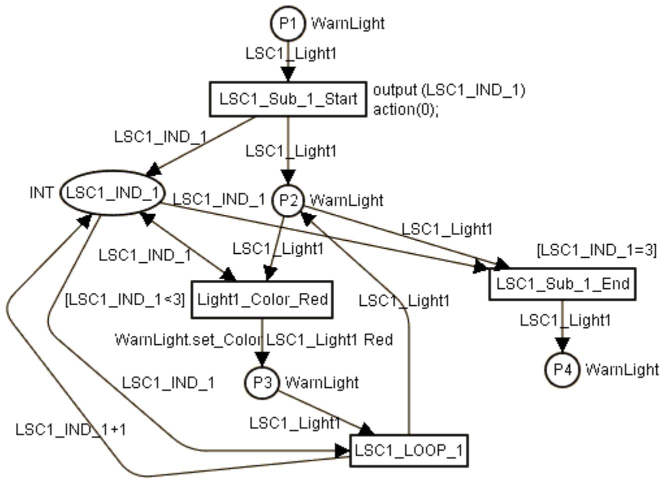


Figure 3.15: CPN model for a *fixed* loop

If a *fixed* loop uses a variable to determine the numbers of iterations, the steps to be followed in order to obtain the CPN model are similar to the steps applied to a *fixed* loop, which uses a constant to determine the number of iterations, except:

- at guard conditions, *max* is the variable used to determine the number of iterations;
- transitions are input and output of the place created to represent the variable, as described in Section 3.3, when dealing with symbolic messages.

The mapping process for a *dynamic* loop is the same applied to a *fixed* loop, but the number of iterations is undefined.

In a case of an *unbounded* loop, it should:

- model all constructions presented in the loop's scenario, according to their mapping rules presented earlier;
- after modeling all constructions, create a transition, which represents returning to the beginning of the loop's scenario (a new iteration). The label of this transition is formed by the chart's name, followed by "_LOOP_ID", where *ID* is equals to the subchart *ID*.

As an example, Figure 3.16 shows the CPN model obtained for the *unbounded* loop presented in Figure 2.13(c). The process is similar to a *fixed* loop. In this case, a controlling condition, represented by transitions *LSC2_CD_1_TRUE* and *LSC2_CD_1_FALSE*, decides if the loop should be iterated. Loop's scenario is abandoned when transition *LSC2_CD_1_FALSE* fires.

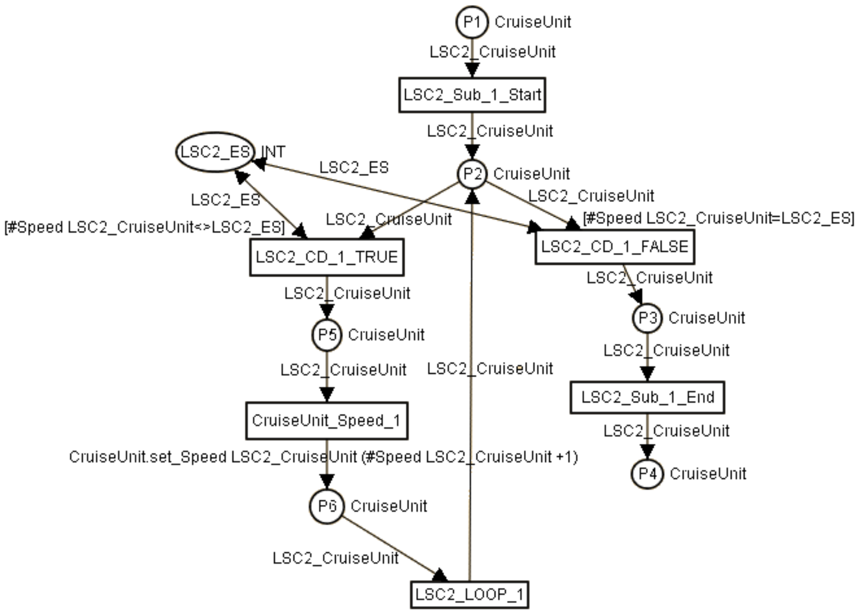


Figure 3.16: CPN model for an *unbounded* loop

## 3.9 Time Restrictions

LSC language allows to establish time restrictions for real-time systems.

In order to obtain the CPN model for a time restriction, the following steps must be taken:

1. map the constructions (assignments and conditions), which are participating in the time restriction, according to the rules presented earlier (Section 3.5 and Section 3.6);

2. create a *timed* type instead of untimed one to represent the instance (Section 3.1). This type allows to apply a *time stamp* for a *token*, i.e., each *token* can have a time associated with it. This *time stamp* indicates when the *token* is available;

3. assign to the arc inscription that leaves the transition that represents the assignment to the place that represents the affected variable in the assignment, the expression *IntInf.toInt(time())*, where *IntInf.toInt* is a CPN ML function for converting the time to an integer number and *time()* returns the actual time (global time);

4. if there is a minimum delay, then after modeling the first condition, assign the following guard conditions *[IntInf.toInt(time())* $>$ *VarAsg+Min-Delay]* and *[IntInf.toInt(time())* $<=$ *VarAsg+Min-Delay]* to the transitions that represents a true and false value, respectively, where *IntInf.toInt(time())* returns an integer that represents the actual time, *VarAsg* is the variable that stores the time in the assignment and *Min-Delay* is the minimum delay. Assign to the arc inscription that leaves the transition that represents the false value, the expression *InstVar @+1*, where *InstVar* is the variable that represents the instance;

5. if there is a maximum delay, then after modeling the second condition, assign the following guard conditions *[IntInf.toInt(time())* $<=$ *VarAsg+Max-Delay]* and *[IntInf.toInt(time())* $>$ *VarAsg+Max-Delay]* to the transitions that represents a true and false value, respectively, where *IntInf.toInt(time())* returns an integer that represents the actual time, *VarAsg* is the variable that stores the time in the assignment construction and *Max-Delay* is the maximum delay.

As a time restriction is formed by assignments and conditions, the formalism behind that is the same presented when mapping assignments and conditions, with the addition of the following formal definition.

**Definition 3.9.1.** *(Function for mapping a time restriction construction) Let* $a_i \in A_L$ *be an assignment of chart L,* $c_i, c_{i+1} \in C_L$ *be conditions of chart L,* $I_{c_i} \subseteq I_L$ *be a set of instances that synchronizes with the condition* $c_i$*,* $I_{c_{i+1}} \subseteq I_L$ *be a set of instances that synchronizes with the condition* $c_{i+1}$*,* $i_i \in I_{c_i}$ *be an instance that synchronizes with the condition* $c_i$ *and* $i_{i+1} \in I_{c_{i+1}}$ *be an instance that synchronizes with the condition* $c_{i+1}$*. The function* $MTS : A_L \times C_L \mapsto N$ *maps a time restriction to a CPN model, where* $a_i.Timed = True \wedge c_i.Timed = True \vee c_{i+1}.Timed = True$*.* $N_i \in N$ *is defined as follows:*

- *if exists a condition that establishes a minimal delay, then:*
  - $G(t_{ii})$ *="*$IntInf.toInt(time())$ $Oper$ $MV(a_i.V)$ $+MinDelay(c_i)$*",* $Oper$ *can be* $>$ *or* $>=$*;*

  - $G(t_{ii+1})$ = "$IntInf.toInt(time())$ $Oper$ $MV(a_i.V)$ + $MinDelay(c_i)$", $Oper$ can be $<$ or $<=$;
  - $E((t_{ii+1}, p_{ii}))$ ="$M_O(i_i)@+1$";

- *if exists a condition that establishes a maximal delay, then:*
  - $G(t_{ii+2})$ = "$IntInf.toInt(time())$ $Oper$ $MV(a_i.V)$ + $MaxDelay(c_{i+1})$", $Oper$ can be $<$ or $<=$;
  - $G(t_{ii+3})$ = "$IntInf.toInt(time())$ $Oper$ $MV(a_i.V)$ + $MaxDelay(c_{i+1})$", $Oper$ can be $>$ or $>=$,

  *where $MinDelay$ and $MaxDelay$ are functions that return the minimal delay and the maximal delay of the timed condition, respectively.*

Figure 3.17 shows the CPN model for instance *O2*, considering the time restriction imposed between events *M1()* and *M2()* presented in scenario of Figure 2.14. Transitions *VertDel_CD_1_TRUE* and *VertDel_CD_1_FALSE* represent the first condition of the time restriction, which imposes the minimum delay. In this case the transition *VertDel_CD_1_FALSE* fires until the guard condition assigned to transition *VertDel_CD_1_TRUE* is not satisfied. This represents the semantics of a *hot* condition that should be continually evaluated until returns true. Transitions *VertDel_CD_2_TRUE* and *VertDel_CD_2_FALSE* represent the second condition of the time restriction. One of the two transitions should fire, when the execution arrives at this point. If the transition *VertDel_CD_2_TRUE* fires, then the time restriction imposed between the events was satisfied. On the other hand, if transition *VertDel_CD_2_FALSE* fires, then this indicates that the time restriction was not respected, indicating a requirement violation.

## 3.10 Time Events

Reactive real-time systems are often required to react to the passage of time, and not only to refer to it when constraining the timing of other events of interest. In order to express such requirements, generally termed *time events*, a special object instance representing the clock is available (Clock Instance), and it can be added to the LSC scenarios. Within this instance one can refer to the *Tick* event, which represents an actual clock, i.e., the passage of a single time unit. This event can be placed in a prechart to trigger desired actions, or in the main chart, thus explicitly forcing delays.

When a *Tick* event occurs, it is fully unified when it is enabled in all charts it is available, in other case these events are seen as different events,

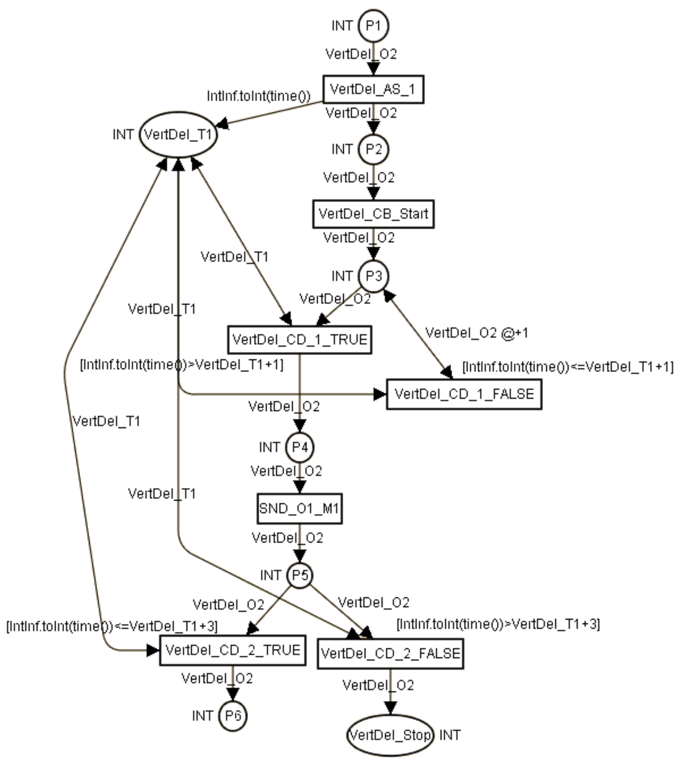


Figure 3.17: CPN model for a *Vertical Delay* time restriction

so these *time events* will take place at different time, in an appropriate moment.

When obtaining the corresponding CPN model of a *Tick* event, it should be seen as a simple LSC message with an internal object as sender, with the following additional steps:

- the label of the obtained transition is formed by "TICK_ID", where *ID* is a unique integer number;
- assign to the arc inscription that leaves the transition, the expression *InstVar @+1*, where *InstVar* is the variable that represents the instance *Clock*. This step represents the passage of one time unit.

## 3.11 Joining LSC Constructions

In the previous sections, it was presented some steps on how to obtain a CPN model which represents individual LSC constructions. After mapping these individual constructions, the obtained CPN models must be joined in order to find a final CPN model that represent the LSC specification. So, the following steps must be applied to map an LSC chart:

- obtain an individual model for each instance inside a chart, where the constructions available in the instance line should be mapped from top to bottom, following the corresponding rules that were presented.

  Let $I_L$ be the set of instances of chart $L$, $i_i, i_{i+1}, ..., i_m \in I_L$ be instances of chart $L$, $l_j^i, l_{j+1}^i, ..., l_n^i$ be locations of instance $i_i$, where $l_j^i$ precedes $l_{j+1}^i$, and so on, therefore $l_j^i < L\ l_{j+1}^i, ..., l_{n-1}^i < L\ l_n^i$, as defined in the execution order statement described in Section 3.3. For $i_i, i_{i+1}, ..., i_m$, apply the mapping rules for each location $l_j^i, l_{j+1}^i, ..., l_n^i$, where $l_j^i$ is the first location to be mapped and $l_n^i$ is the last, according to the execution order described in Section 3.3;

- the output place of a transition of a CPN model that represents an LSC construction, must be the input place of the transition of the CPN model that represents the next construction to be mapped.

  Let $N_j, N_{j+1}, ..., N_n$ be the corresponding CPN models obtained for the locations $l_j^i, l_{j+1}^i, ..., l_n^i$ of instance $i_i$, $N_j.P$ be the set of places of $N_j$, $N_j.T$ be the set of transitions of $N_j$, $N_j.A$ be the set of arcs, $N_j.E$ be an arc function that assigns inscriptions to arcs. $p_k \in N_j.P$ is a place and $t_l \in N_j.T$ is a transition, where there is an arc $(t_l, p_k) \in N_j.A$ and $N_j.E(t_l, p_k)$ is equals to $M_O(i_i)$. $p_r \in N_{j+1}.P$ is a place and $t_s \in N_{j+1}.T$ is a transition, where there is an arc $(p_r, t_s) \in N_{j+1}.A$ and $N_{j+1}.E(p_r, t_s)$ is equals to $M_O(i_i)$. Places $p_k$ and $p_r$ must be united according to these conditions, by applying the label function to a place ($LABP$), to rename $p_r$ to $p_k$, therefore, $N_j.A = N_j.A \cup \{(p_r, t_s)\}$ and $N_{j+1}.A = N_{j+1}.A - \{(p_r, t_s)\}$. $N_j, N_{j+1}, ..., N_n$ models must be joined according to these rules;

- after all individual models (instances inside of a chart) were found, transitions of same label should be merged, as well as the places that possess the same label. The inputs and outputs of these transitions and places that should be merged, are joined in the final obtained model (CPN model that represents the chart).

A system specification contains more than one chart, and some messages may be specified in more than one chart. In order to join the CPN models that represent each individual chart, all transitions with same label and all places with same labels must be merged, as described earlier.

## 3.12 Comparing Petri Net and LSC Semantics

This section considers a small example in which some of LSC constructions are adopted, to compare the labeled transition system directly generated from LSC specifications and the occurrence graph generated from the respective CPN models. Therefore, this section shows that both models (LSC and the respective CPN) lead to bisimilar [43] transition systems. It is important to stress, however, that these transition systems are not isomorphic, since the transformation of LSC to CPN inserts transitions that do not correspond to LSC visible events (messages), but are necessary for representing the correct net's structure.

The adopted approach consists of four phases, as depicted in Figure 3.18.

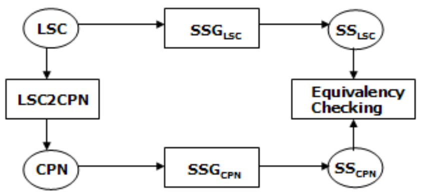


Figure 3.18: Validation phases

Initially, the adopted methodology considers the CPN model that represents an LSC chart, obtained by executing LSC2CPN engine in order to obtain the corresponding state space ($SS_{CPN}$) of a Coloured Petri Net. This process ($SSG_{CPN}$) can be automatically achieved in many available CPN tools.

The LSC State Space Generation ($SSG_{LSC}$) obtains a state space of an LSC chart, in which, each state presents the current configuration of all instances inside an LSC chart and the enabled events at this point.

In order to check if the methodology presented in this work generates faithful models, the comparison process should verify whether the obtained LSC State Space ($SS_{LSC}$) and the obtained CPN State Space ($SS_{CPN}$) are equivalent or not.

Next, it is presented a toy example, in order to demonstrate how this comparison process should be applied. Take a look at the LSC scenario shown in Figure 3.19, in which it is presented a receiving calling scenario. In this scenario, whenever *Chan1* (Channel1) sends a calling request to the *Switch*, then the *Switch* forwards this request to *Chan2* (Channel2). If *Chan2* is not in order (ready), then it informs the *Switch* that the calling

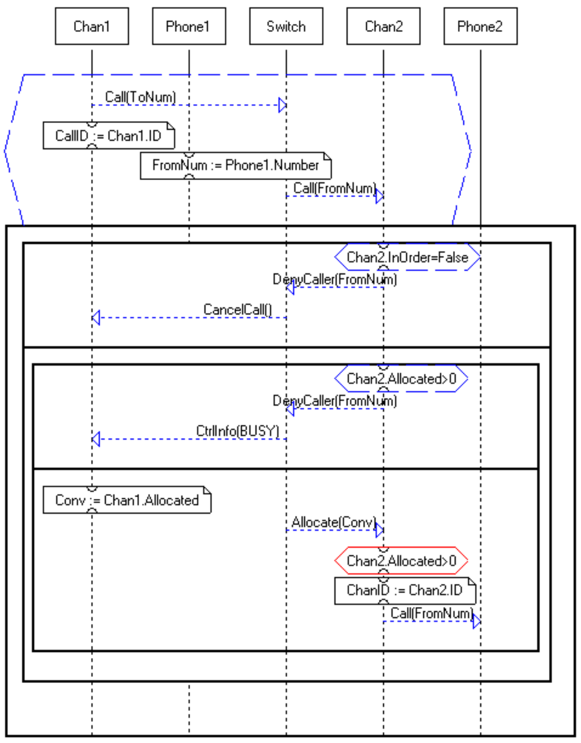
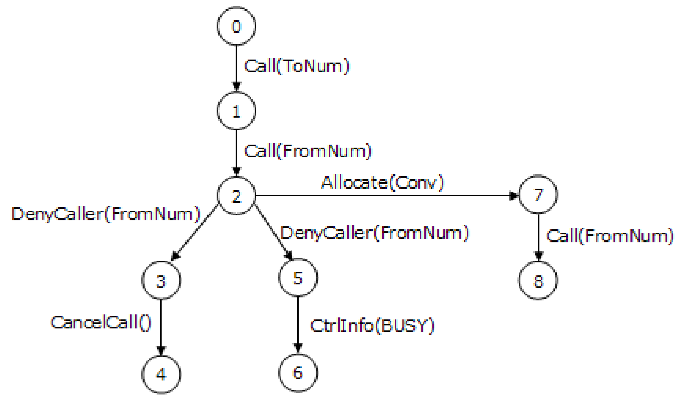


Figure 3.19: LSC scenario

can not be made, and the *Switch* tells *Chan1* that the calling was canceled. If *Chan2* is in order but it is allocated, then the calling is denied and *Chan1* is informed that *Chan2* is busy, otherwise (not allocated) *Chan2* is allocated and the communication is established between *Phone1* and *Phone2*.

When applying $SSG_{LSC}$ phase, an LSC state is represented by the current value of all instances' properties, variables values, and enabled events at this point. A state changing takes place whenever firing one of these enabled events. Figure 3.20 presents the reachability graph of the LSC scenario presented in Figure 3.19, which it is obtained through an exhaustive simulation in the *Play-Engine* tool.

By applying the corresponding mapping steps in the LSC scenario presented in Figure 3.19 as described in Chapter 3, Figure 3.21, Figure 3.22, Figure 3.23, Figure 3.24 and Figure 3.25 depict the obtained CPN model for *Chan1*, *Phone1*, *Switch*, *Chan2* and *Phone2*, respectively. In order to obtain the final CPN model, these individuals CPN model must be joined, where transitions with same label must be merged, in which, input and output places of one transition are joined with input and output places of


$$\begin{tikzcd}
 & & 0 \arrow[d, "Call(ToNum)"] & \\
 & & 1 \arrow[d, "Call(FromNum)"] & \\
 & & 2 \arrow[dl, "DenyCaller(FromNum)"'] \arrow[dr, "DenyCaller(FromNum)"] \arrow[rr, "Allocate(Conv)"] & & 7 \arrow[d, "Call(FromNum)"] \\
 & 3 \arrow[d, "CancelCall()"'] & & 5 \arrow[d, "CtrlInfo(BUSY)"] & 8 \\
 & 4 & & 6 &
\end{tikzcd}$$

```
0
[{Chad1.ID=1,Chad1.Allocated=0,Chan1.InOrder=undefined,Chad2.ID=2,Chad2.Allocated
=0,Chad2.InOrder=undefined,Phone1.Number=1111,ToNum=0,CallID=0,ChanID=0,From
Num=0,Conv=0},{Call(ToNum)}]
1
[{Chad1.ID=1,Chad1.Allocated=0,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=0,C
had2.InOrder=undefined,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=0,FromNum
=1111,Conv=0},{Call(FromNum)}]
2
[{Chad1.ID=1,Chad1.Allocated=1,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=0,C
had2.InOrder=undefined,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=0,FromNum
=1111,Conv=0},{DenyCaller(FromNum),Allocate(Conv)}]
3
[{Chad1.ID=1,Chad1.Allocated=1,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=0,C
had2.InOrder=False,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=0,FromNum=111
1,Conv=0},{CancelCall()}]
4
[{Chad1.ID=1,Chad1.Allocated=1,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=0,C
had2.InOrder=False,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=0,FromNum=111
1,Conv=0},{empty}]
5
[{Chad1.ID=1,Chad1.Allocated=1,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=1,C
had2.InOrder=True,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=0,FromNum=111
1,Conv=0},{CtrlInfo(BUSY)}]
6
[{Chad1.ID=1,Chad1.Allocated=1,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=1,C
had2.InOrder=True,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=0,FromNum=111
1,Conv=0},{empty}]
7
[{Chad1.ID=1,Chad1.Allocated=1,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=0,C
had2.InOrder=True,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=0,FromNum=111
1,Conv=1},{Call(FromNum)}]
8
[{Chad1.ID=1,Chad1.Allocated=1,Chan1.InOrder=True,Chad2.ID=2,Chad2.Allocated=1,C
had2.InOrder=True,Phone1.Number=1111,ToNum=2,CallID=1,ChanID=2,FromNum=111
1,Conv=1},{empty}]
```

Figure 3.20: Reachability graph of the LSC scenario presented in Figure 3.19

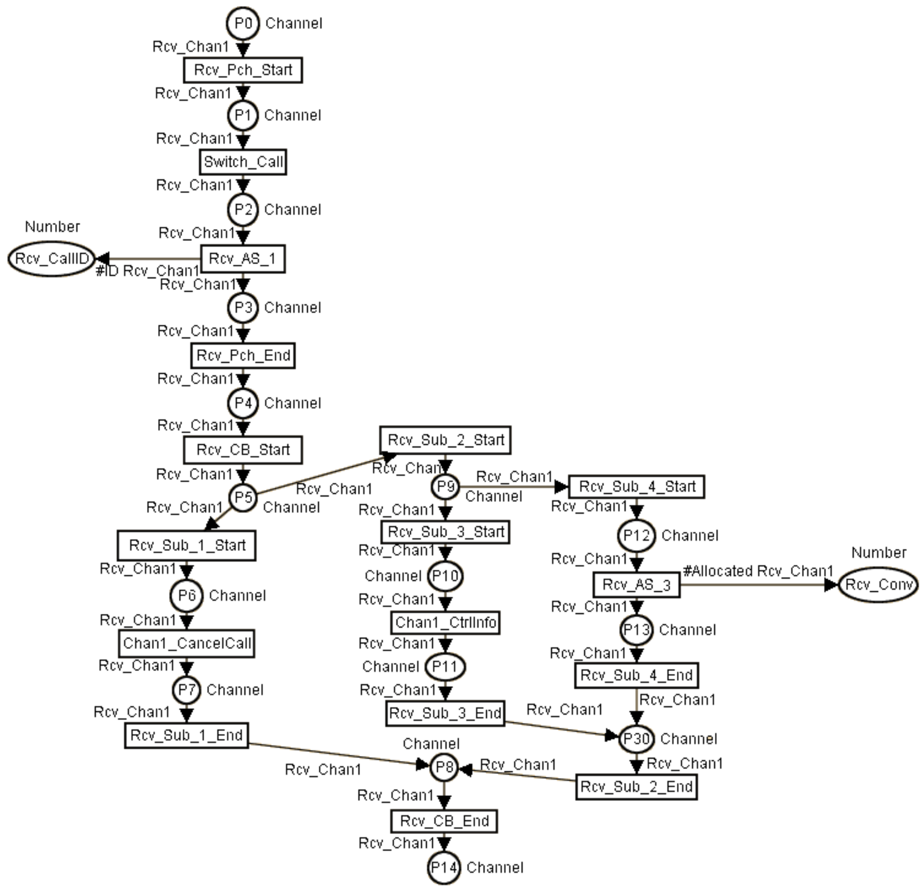


Figure 3.21: CPN model for *Chan1* instance

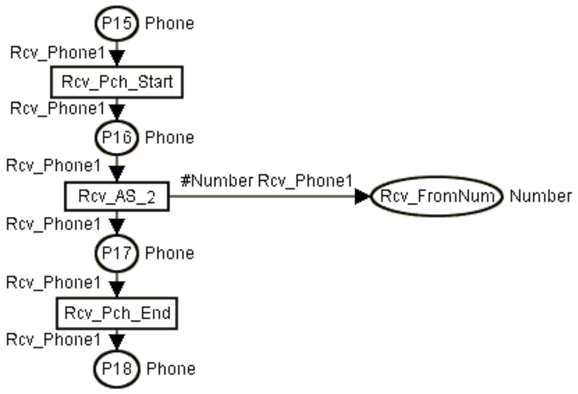


Figure 3.22: CPN model for *Phone1* instance

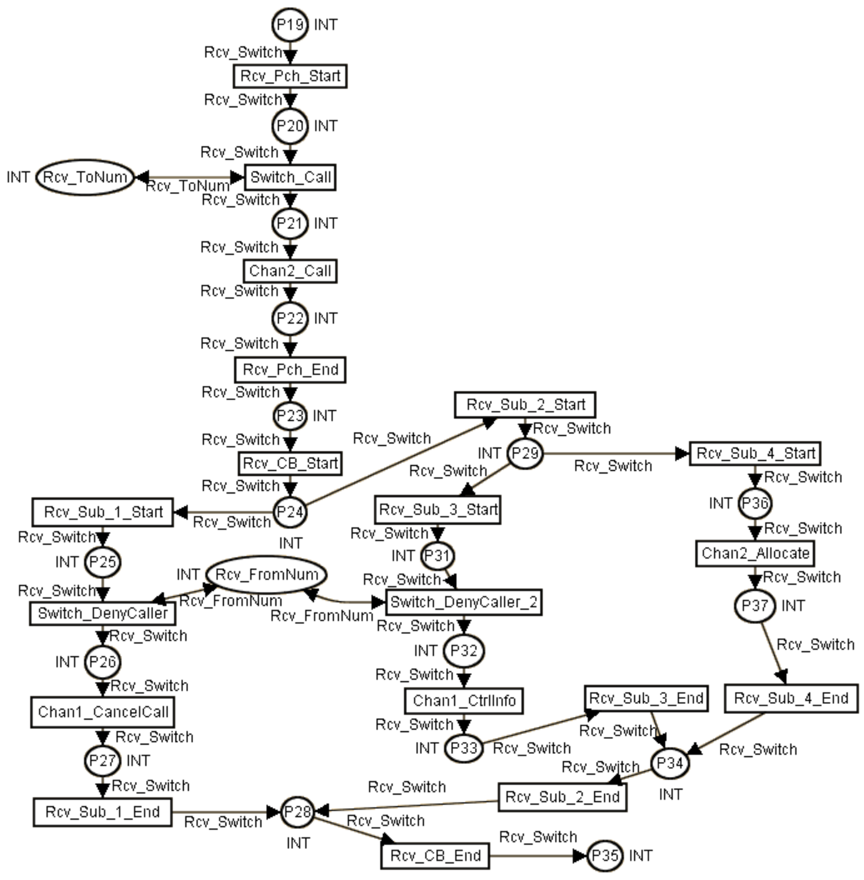


Figure 3.23: CPN model for *Switch* instance

the other transition. The same is valid to places with same label.

The CPN Tools is adopted for creating the CPN model, as well as, for finding the corresponding reachability graph. The reachability graph of the CPN model is shown in Figure 3.26. Each state is represented by places which have at least one token.

The mapping process takes into account “visible” events (messages) and “hidden” events (synchronization points, assignments, conditions), therefore the reachability graph of the CPN model presents “visible” states (states that represent “visible” events) and “hidden” states (states that represent “hidden” events). Figure 3.27 presents the reachability graph of the CPN model highlighting the “hidden” states.

The reachability graph of the LSC scenario does not take into account “hidden” events, because until nowadays there is no tool to generate the reachability graph of a LSC scenario, so we built the reachability graph of

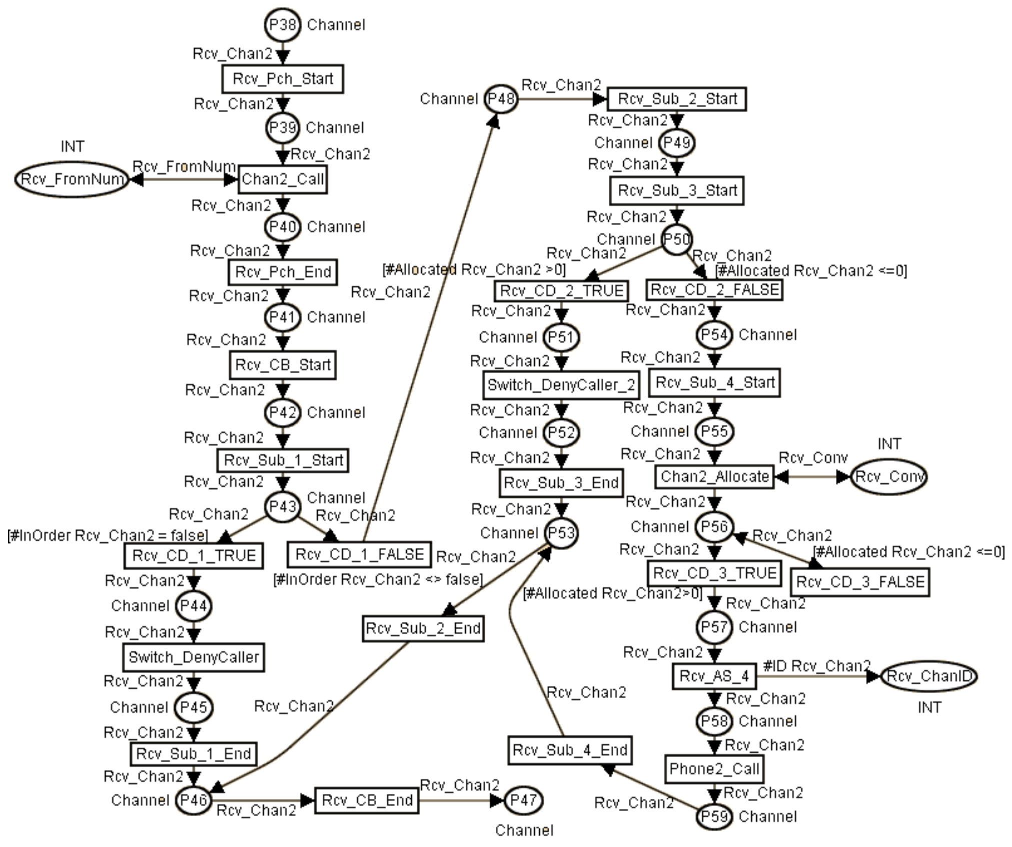


Figure 3.24: CPN model for *Chan2* instance

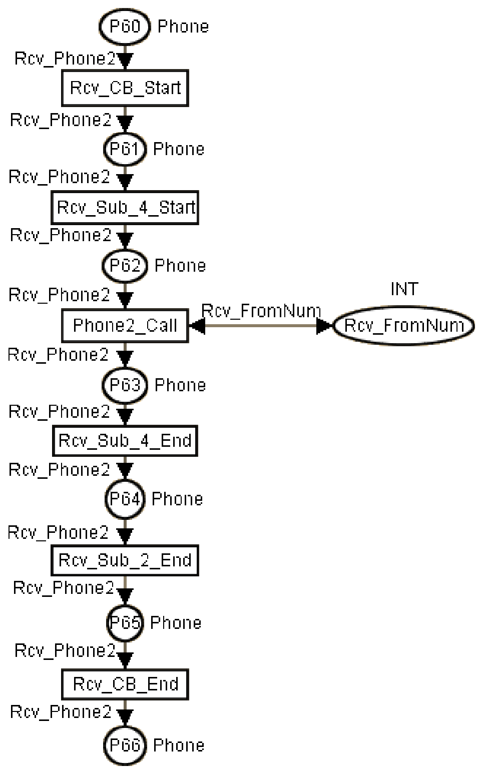
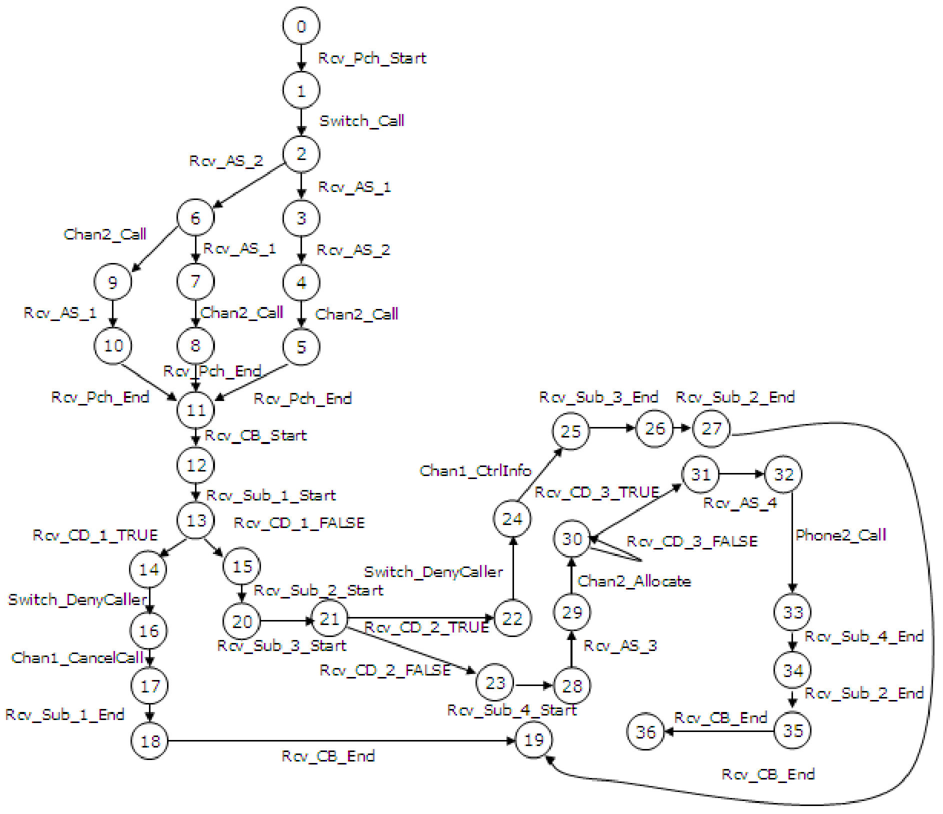


Figure 3.25: CPN model for *Phone2* instance

the presented LSC scenario through simulation at *Play-Engine* tool. Therefore, if we suppress the "hidden" states from the reachability graph of the CPN model, it is obtain the reachability graph presented in Figure 3.28. As we can see, the reachability graph of the CPN model presented in Figure 3.28 is bisimilar to the reachability graph of the LSC scenario presented in Figure 3.20, therefore, the semantics of the LSC scenario presented in Figure 3.19 and of the obtained CPN model are equivalent, i.e., each state transition on the LSC reachability graph (represented by message invocation) has its equivalent state transition on the CPN reachability graph (represented by transition firing).

This simple example shows how CPN models can be compared to LSC scenarios in order to check if they are equivalents. However, the adopted method does not aim, at this point of our research, being complete or even validating the mapping method. Nevertheless, this approach has been applied in many other case studies, in order to check if LSC and CPN automatic generated labeled transition systems are equivalent or not.



| | |
|---|---|
| **0** | P0, P15, P19, P38, P60, Rcv_ToNum |
| **1** | P1, P16, P20, P39, P60, Rcv_ToNum |
| **2** | P2, P16, P21, P39, P60, Rcv_ToNum |
| **3** | P3, P16, P21, P39, P60, Rcv_ToNum, Rcv_CallID |
| **4** | P3, P17, P21, P39, P60, Rcv_ToNum, Rcv_CallID, Rcv_FromNum |
| **5** | P3, P17, P22, P40, P60, Rcv_ToNum, Rcv_CallID, Rcv_FromNum |
| **6** | P2, P17, P21, P39, P60, Rcv_ToNum, Rcv_FromNum |
| **7** | P3, P17, P21, P39, P60, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **8** | P3, P17, P22, P40, P60, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **9** | P2, P17, P22, P40, P60, Rcv_ToNum, Rcv_FromNum |
| **10** | P3, P17, P22, P40, P60, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **11** | P4, P18, P23, P41, P60, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **12** | P5, P18, P24, P42, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **13** | P6, P18, P25, P43, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **14** | P6, P18, P25, P44, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **15** | P6, P18, P25, P48, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **16** | P6, P18, P26, P45, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **17** | P7, P18, P27, P45, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **18** | P8, P18, P28, P46, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **19** | P14, P18, P35, P47, P66, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **20** | P9, P18, P29, P49, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **21** | P10, P18, P31, P50, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **22** | P10, P18, P31, P51, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **23** | P10, P18, P31, P54, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **24** | P10, P18, P32, P52, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **25** | P11, P18, P33, P52, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **26** | P30, P18, P34, P53, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **27** | P8, P18, P28, P46, P61, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **28** | P12, P18, P36, P55, P62, Rcv_ToNum, Rcv_FromNum, Rcv_CallID |
| **29** | P13, P18, P36, P55, P62, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv |
| **30** | P13, P18, P37, P56, P62, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv |
| **31** | P13, P18, P37, P57, P62, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv |
| **32** | P13, P18, P37, P58, P62, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv, Rcv_ChanID |
| **33** | P13, P18, P37, P59, P63, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv, Rcv_ChanID |
| **34** | P30, P18, P34, P53, P64, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv, Rcv_ChanID |
| **35** | P8, P18, P28, P46, P65, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv, Rcv_ChanID |
| **36** | P14, P18, P35, P47, P66, Rcv_ToNum, Rcv_FromNum, Rcv_CallID, Rcv_Conv, Rcv_ChanID |

Figure 3.26: Reachability graph of the CPN model

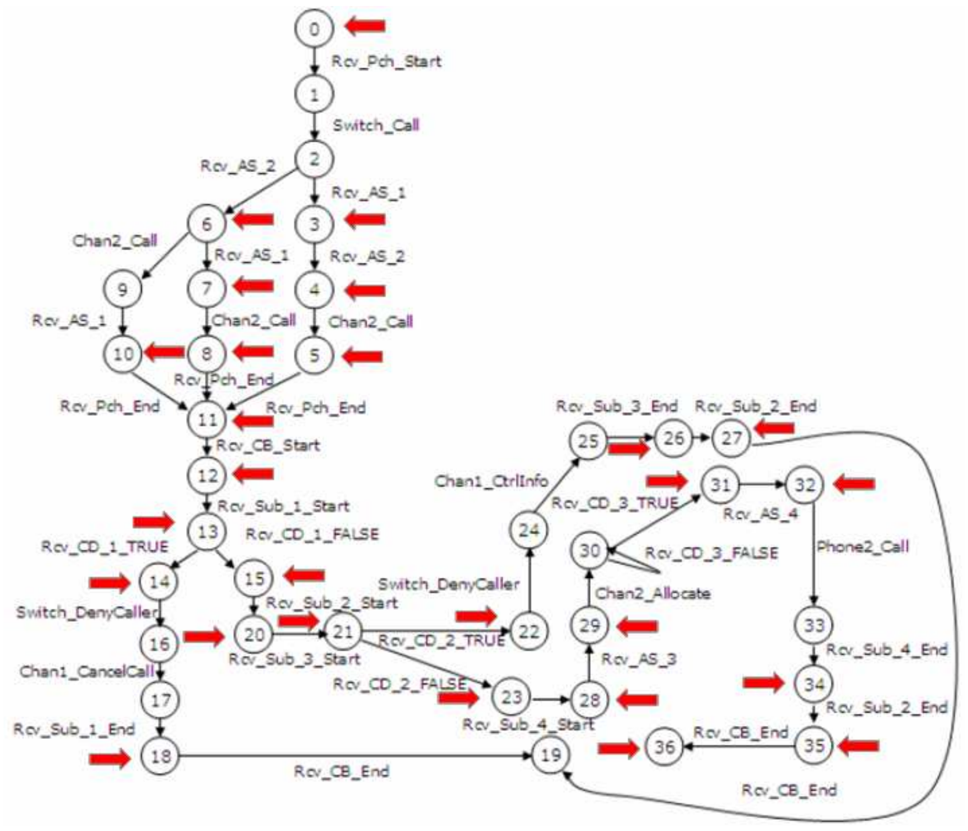


Figure 3.27: “Hidden” states of reachability graph of the CPN model

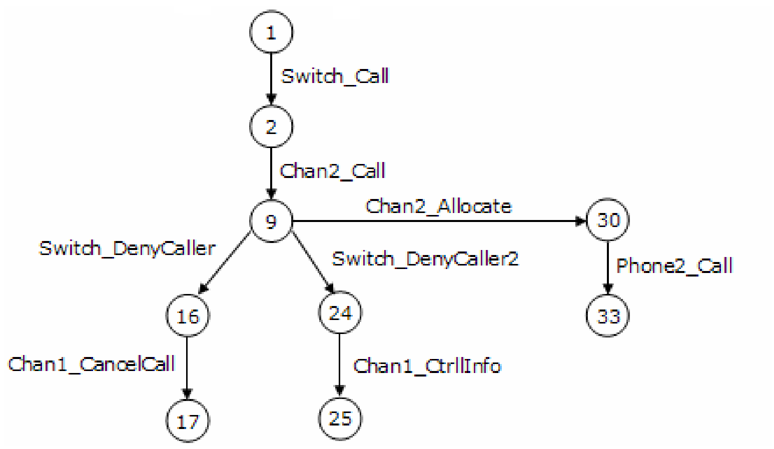


Figure 3.28: Reachability graph of the CPN model without the “hidden” states

## 3.13 Concluding Remarks

In this chapter we presented the mapping process, describing how to obtain an equivalent CPN model for each LSC construction presented in the previous chapter. The mapping process was explained in the following way: the steps that must be followed are described, then it is presented the formalism and the corresponding CPN model, and finally an example is shown.

Once the individual CPN models were obtained, we described the steps that must be followed in order to obtain a final CPN, formed by joining these individual models, in which transitions and places with same label must be joined where their inputs and output are united.

At the end of the chapter we presented an approach to compare the LSC and CPN semantics, which was applied in a specific example. This approach can not be considered as a general validation process, however we applied the described approach in few case studies to verify if the obtained CPN model is equivalent to the mapped LSC specification.

# Chapter 4

# Case Studies

*This chapter presents two case studies, in which the mapping process is applied and some interesting properties are verified through custom queries. Also it is made an analysis of the obtained CPN models.*

## 4.1 Pulse Oximeter

In order to show the practical usability of the proposed mapping process, a pulse-oximeter [30] has been considered as a case study. This electronic equipment is responsible for measuring the blood oxygen saturation using a non-invasive method. This equipment is widely used in critical care units (CCU).

The pulse-oximeter was described using the following scenarios: *Excitement*, *Cross-section* and *Management*.

The *Excitement* scenario (see Figure 4.1) specifies that if a *Cross-section* operation is taking place, then the channel is read from digital-analogic conversor (*DAConv*), the data is processed by the processor (*Proc*) and it is sent to *Management* module to present the information at the interface (*Display*), otherwise the sign is adjusted.

The *Cross-section* scenario (see Figure 4.2) specifies that whenever the sigh calibration takes place, then the processor should emit red and infrared pulses interchanged.

The *Management* scenario (see Figure 4.3) starts the *Cross-section* process and presents the obtained information on the interface (*Display*).

This work presents some results related to properties analysis and verification.

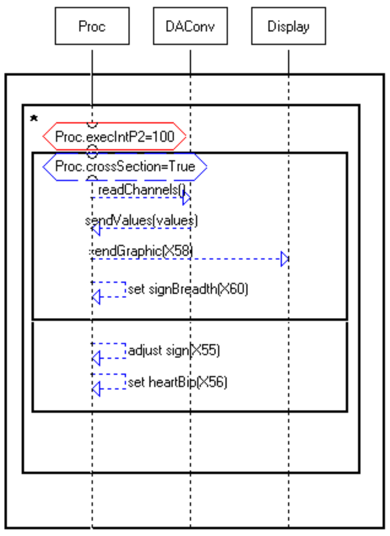


Figure 4.1: Excitement scenario

In order to obtain the CPN model that represents the scenarios presented in Figure 4.1, Figure 4.2 and Figure 4.3, we follow the steps presented next.

We obtain the individual model for each one of the instances in the chart, then for each instance line, the elements should be mapped from top to bottom, following the corresponding rules that were presented. The transition that represents the beginning of a chart must be an input of each place that represents a variable used in the LSC chart, where the formed arc inscriptions initialize each place with one token. The output place of a transition, that represents an LSC construction (message, condition, assignment, etc), must be the input place for the transition that represents the next element to be mapped. After all individual models were found, transitions with same label should be put upon, as well as the places that possess the same label. The inputs and outputs of these transitions and places that should be put upon, will be joined in the final model. Places representing each refereed variable in the chart should be an input to the transition that represents the chart body ending.

When LSC2CPN engine finishes translating the LSC scenarios, CPN Tools [1] could be used to analyse and verify properties of the specified system.

After a basic analysis of the obtained CPN model, it is observed that the specified system has the following properties:

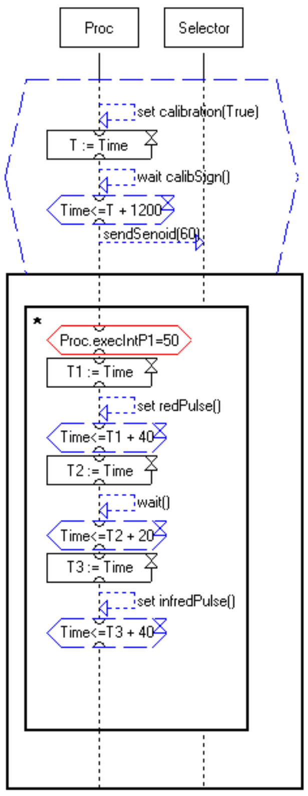


Figure 4.2: Cross-section scenario

**Deadlock Freedom** The analysis of pulse-oximeter system indicates that the system is not deadlock free, due to the presence of "good" deadlocks, which is not bad because there are some pre-conditions that should be satisfied in order to blood measure takes place by the pulse-oximeter equipment. Therefore, the pulse-oximeter system executes on expected conditions, as it was described at the LSC specification. On the other hand, if the *time stamp* (time constraint) of the transition that represents "*set redPulse()*" message (Cross-section scenario) is modified to be larger than 40 time units, then when analysing the CPN model it is found a reachable deadlock state ("bad" deadlock), since a time restriction is not respected, as it was specified (see Figure 4.2), the "*set redPulse()*" message can not take more than 40 time units to execute, so in this case as it was expected, the system is not deadlock free;

**Liveness** The CPN model of pulse-oximeter system is not *live*, because

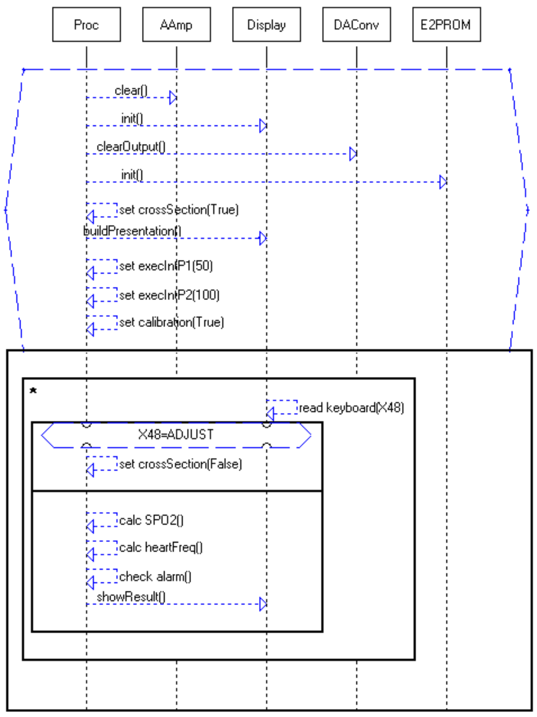


Figure 4.3: Management scenario

there are situations, in which the system is lead to a "good" deadlock state. If these situations are unconsidered as well as a set of transitions representing activities that are only executed during start-up phase, the pulse-oximeter system can be considered a *live* one. Through *liveness* property, it can be verified if the pulse-oximeter equipment is prepared to measure the blood oxygen, once a pre-condition have been satisfied, until the system is shut down;

**Boudedness** The pulse-oximeter system is a *safe* system, because for all reachable markings the places have at most one token. So, none of the possible sequence of events can lead the pulse-oximeter system to an unpredictable state;

**Reversible** The system is not reversible to its initial state, but there are *home states*. The pulse-oximeter system can not returns to its initial state after starting, so the initial state can only be reached if the system is shut down and restarted, however there are some markings (states) that can be reached again by firing a sequence of transitions (events);

**Bounded-Fairness** Once the execution of pulse-oximeter is inside char body scenarios, the firing sequence (sequence of events) is *unconditionally fair*. So, as it is expected, every time the pulse-oximeter system enters inside char body scenarios, it will always execute the same sequence of steps;

**Conservativeness** A Petri net covered by place invariants is *conservative*, that is, the CPN model for pulse-oximeter system is not *conservative*, since places describing pre-conditions within precharts are not covered by place invariants. However, if the precharts are unconsidered, the CPN model is *conservative*, which is an interesting property for embedded system design. Therefore, besides *boundedness*, which depicted that analysed system does not generate an infinite number of states, *conservativeness* shows that the respective specification also does not consume resources without further liberating them.

Besides the basic analysis, some specific properties can be verified, as it is shown next.

In the Excitement scenario (see Figure 4.1), the two underlying subcharts, with a guard condition (*Proc.crossSection=True*) at the top of the first one, represent an *if-then-else* construction. The events in the scope of the first subchart are executed if the guard condition is evaluated to a true value. If this condition is evaluated to false, the events of the second subcharts are executed. If this condition changes its value when executing the events in the first subchart, then occurs a requirement violation and this subchart must be aborted. Therefore, it can be verified if it is possible that some event of this first subchart can be executed when the user presses the adjust button (see Figure 4.1). This property can be verified by using the query language ASK-CTL [1]. The following formula checks this property: FORALL_UNTIL (TT, AND ( AF( “Events”, AreEventsEnabled), NF(“CrossSection”, IsCrossSection))). AreEventsEnabled and IsCrossSection are CPN ML [1] functions that check if any event of the first subchart is enabled and if the device is in cross-section mode, respectively. Applying this formula, a false value is returned, so it indicates that it is not possible to execute some event of the first subchart (Management scenario), when the user presses the adjust button. *NF* is a node function, where its arguments are a string and a function which takes a state space node and returns a boolean. The string is used when an ASK-CTL formula evaluates as false in the model checker. In this case the model checker shows a diagnostic message explaining why the formula is false, using the string in the message. *AF* is the arc function and is analogous to *NF*, only that it is a transition formula and thus only makes sense to use as a transition sub-formula.

As described in [30], the excitement scenario has a priority over cross-section scenario, so when executing the events in excitement scenario, none of the events of cross-section scenario can be executed. This property can be checked through the following formula: FORALL_UNTIL (TT, AND( AF("Excitement", AreExcitementEventsEnabled), AF("Cross-section", Are-CrossSectionEventsEnabled))). This formula asks if there is a node in the state space where some event of excitement scenario is enabled to fire at the same time that some event of cross-section scenario can be executed. Applying this formula, a true value is returned, hence there is a requirement violation in this specification, since the specification was built in an incorrect way and this is verified when evaluating this formula.

State space queries [1] can be used as another approach to properties verification. The state space query SearchNodes( EntireGraph, CanSendRedPulse, NoLimit, $fn\ n => n$,[],op::) verifies if a red pulse event (see Figure 4.1) can be sent before setting the calibration to on (see Figure 4.2). *CanSendRedPulse* is a CPN ML function that checks if the transition that represents the event "set redPulse()" is enabled to fire when the calibration is off (before firing the transition that represents the event "set calibration(True)"). When applying this state space query in the obtained CPN model, it returns nothing, so it is guaranteed that the red pulse is always sent after setting the calibration to on.

## 4.2 ConnectOK

*ConnectOK* is a portable and mobile data terminal with an internal GSM/GPRS communication module, which can be considered for several tasks, such as reading water, energy and gas consumption. This device is composed by a terminal for reading consumption data and a server to process received data, and and is divided in four management modules:

**Configuration Module** defines some parameters for the terminal data application;

**Communication Module** interacts with a server using a proprietary communication protocol;

**Battery Manager** controls the battery consumption;

**Interface Module** interacts with the user.

This case study applies to Battery Manager module.

The battery management module was specified and its LSC scenarios are presented from Figure 4.4 to Figure 4.10. Next a brief description is presented detailing what each of these scenarios performs.

**Scenario 1** Whenever the menu options is presented (Interface Module), if more than ten time unit have been passed without an user interaction, then the micro-controller sounds a beep and shuts down the terminal;

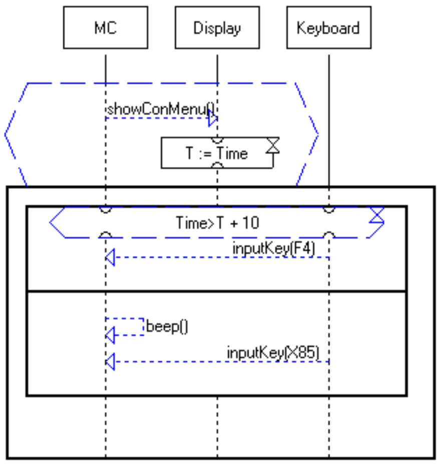


Figure 4.4: Battery Manager Scenario 1

**Scenario 2** If the user presses F2 key, then the micro-controller verifies the battery level and its status is presented in the screen for three time units and then the content menu is presented (Interface Module);

**Scenario 3** If the user presses F3 key, the micro-controller must shut down the terminal;

**Scenario 4** F4 key activates battery loader process. If the loader is not connected, then the terminal must shut down, otherwise battery status is presented while charging;

**Scenario 5** This scenario specifies that whenever the user pushes a key different from a function key (F2,F3,F4), the battery manager modules executes the following procedure. First, the micro-controller checks the battery level, and if it has the minimal level to execute the task or if the loader is connected, then it proceeds. Otherwise, the screen presenting “Turning off...” is shown and the terminal is shut down;

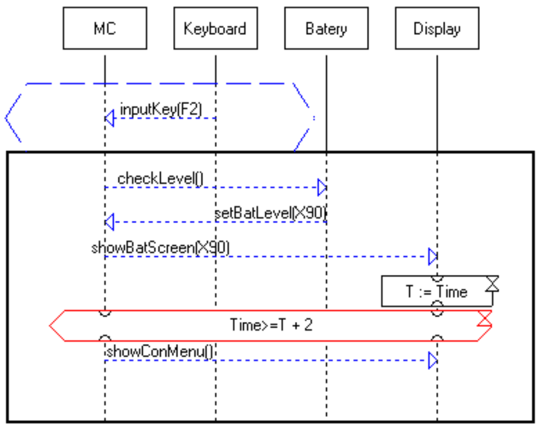


Figure 4.5: Battery Manager Scenario 2

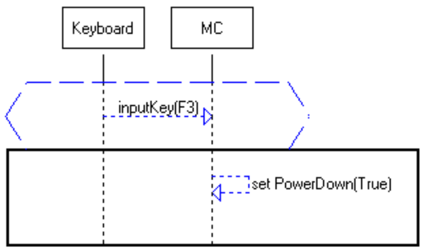


Figure 4.6: Battery Manager Scenario 3

**Scenario 6** If someone connects the battery loader, then the micro-controller is activated and the loader starts charging the battery;

**Scenario 7** Whenever the battery loader is instructed to charge the battery, then the micro-controller continuously checks the battery level until the battery is full, when the battery loader should be turned off.

The procedure described in Section 3.11 should be adopted in order to obtain the CPN model that represents the scenarios presented from Figure 4.4 to Figure 4.10.

After translating the LSC scenarios described previously by applying the LSC2CPN engine, the designer can analyse and verify properties taking into account the obtained CPN model.

After analysing the obtained CPN model, it is observed that the specified system has the following properties:

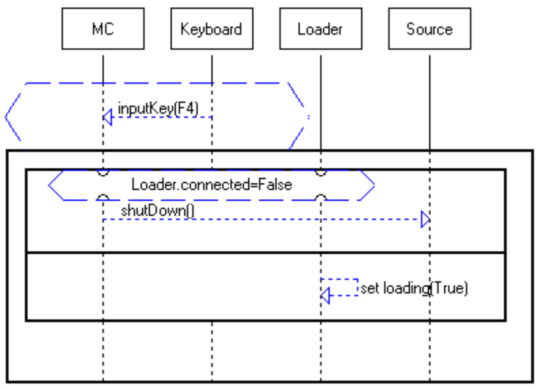


Figure 4.7: Battery Manager Scenario 4

**Deadlock Freedom** The analysis of ConnectOK system indicates that the system is not deadlock free due to the presence of "good" deadlocks, which is not bad because there are some pre-conditions that should be satisfied. Therefore, the ConnectOK system executes on expected conditions;

**Liveness** The CPN model of ConnectOK system is not *live*, because there are situations, in which the system is lead to a "good" deadlock state or the system executes its tasks and finalize;

**Boudedness** The *boudedness* property indicates if the system has predictable states. The ConnectOk system is a *safe* system, because for all reachable markings the places have at most one token. So, none of the possible sequence of events can lead the ConnectOk system to an unpredictable state;

**Reversible** The system is not *reversible* and there is no *home state*, once it is started. The system receives a request from the environment and then executes its tasks and finishes;

**Bounded-Fairness** The *ConnectOk* is not *B-fair*, but it is *unconditionally fair*. It is not *B-fair* since inside chart body scenarios the number of times that a condition is evaluated as true is not *bounded*, comparing to the number of times that a condition is evaluated as false, i.e. the scenario presented in Figure 4.10. And it is *unconditionally fair* because all firing sequences are finite, since the system executes its tasks and terminates;

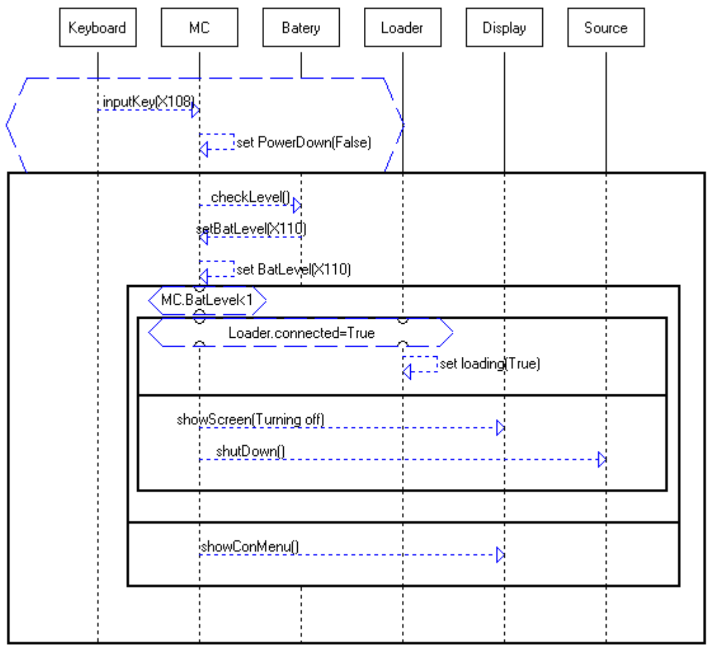


Figure 4.8: Battery Manager Scenario 5

**Conservativeness** The CPN model for ConnectOk system is not *conservative*, since places describing pre-conditions within precharts are not covered by place invariants. However, if the precharts are unconsidered, the CPN model is *conservative*, which is an interesting property for embedded system design. Therefore, besides *boundedness*, which depicted that analysed system does not generate an infinite number of states, *conservativeness* shows that the respective specification also does not consume resources without further liberating them.

After the analysis phase, some system's properties can be verified by constructing queries using the ASKCTL model checking language, as applied in the previous case study. Next, it is presented some interesting requirements of *ConnectOK* system and it is explained how these properties can be verified using model checking formulas.

One important requirement says that the loader must stop battery loading when the battery reaches its maximum charging level (level 5). In order to check this requirement, it must be verified if it is possible to reach a state, in which the *BatLevel* property of *MC* (Micro-controller) has value 5 and the boolean property *loading* of *Loader* has a true value. This property can be checked through the following formula: FORALL_UNTIL (TT,

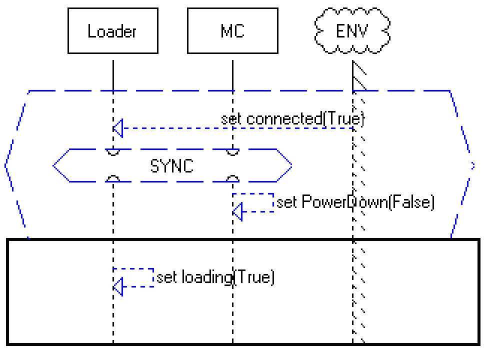


Figure 4.9: Battery Manager Scenario 6

AND( NF("BatteryFull", IsBatteryFull), NF("Loading", IsLoading))), where *IsBatteryFull* and *IsLoading* are functions created using the CPN ML functional language. *NF* is a node function, where its arguments are a string and a function which takes a state space node and returns a boolean. The string is used when an ASK-CTL formula evaluates as false in the model checker. In this case the model checker will print a diagnostic message explaining why the formula is false, using the string in the message. The function *IsBatteryFull* checks if the *BatLevel* property of *MC* has a value 5 (indicates a full battery). If a node satisfies this function, then when applying NF("BatteryFull", IsBatteryFull), it returns true, otherwise returns false. The function *IsLoading* checks if the *loading* property of *Loader* has a *True* value. If a node satisfies this function, then when applying NF("Loading", IsLoading), it returns true, otherwise returns false. The formula, described above, searches all state space for a node, in which NF("BatteryFull", IsBatteryFull) and NF("Loading", IsLoading) return true. By applying this formula in the obtained CPN model, it returns a false value. Therefore it is guaranteed, according to what was specified in the LSC scenarios, that when the battery reaches its maximum charging level, then the loader stops charging the battery. This requirement can be visualized in the scenario of Figure 4.10. Observe the subchart with "*" at the top left corner, which represents an *unbound* loop construction. In this scenario, the battery level is repeatedly checked until reaching level 5 (the condition *MC.BatLevel<5* evaluates as false), when this scenario is abandoned and the next action takes place (*set loading(false)*), which disables the loader charging. It is important to point out that different formulas can be made with the same purpose.

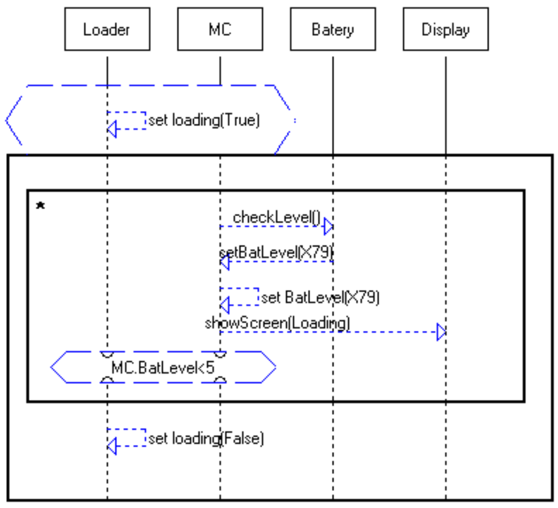


Figure 4.10: Battery Manager Scenario 7

Another important requirement states that the options menu can not be presented if the battery level is lower than the minimum level (level 1) to execute this function (*showConMenu()*). When mapping the presented LSC specification, the obtained CPN model of *showConMenu()* message has one transition and two places (obtained by applying the rules presented earlier). In order to check this requirement, it must be verified if it is possible to reach a state, in which the transition of the CPN model that represents the event *showConMenu()* is enabled to fire while there is a place with a *token* of the *MC* type, in which the value of *BatLevel* property of *MC* is lower than 1 (minimum level). The following formula checks this property: FORALL_UNTIL (TT, AND( AF("ShowConMenu", ShowConMenuEnabled), NF("BelowMinimumLevel", isBelowMinimumLevel))), where *ShowConMenuEnabled* and *isBelowMinimumLevel* are CPN ML functions. *AF* is the arc function and is analogous to *NF*, only that it is a transition formula and thus only makes sense to use as a transition sub-formula. The function *ShowConMenuEnabled* verifies if the transition that represents the event *showConMenu()* is enabled. If a node satisfies this function, then when applying AF("ShowConMenu", ShowConMenuEnabled), it returns true, otherwise it returns false. The function *isBelowMinimumLevel* checks if the *BatLevel* property of *MC* has a value lower than 1. If a node satisfies this function, then when applying NF("BelowMinimumLevel", isBelowMinimumLevel), it returns true, otherwise it returns false. When the formula FORALL_UNTIL (TT, AND( AF("ShowConMenu", ShowConMenu-

Enabled), NF("BelowMinimumLevel", isBelowMinimumLevel))) is applied to the obtained CPN model, a false value is returned, so this requirement was correctly specified in the system's specification, as can be viewed in the scenario of Figure 4.8. In this scenario, there is an *if-then-else* (see Section 3.7) construction, in which, if the controlling condition, at the top of the first subchart evaluates as true, then, if the loader is not connected, the device is shut down. On the other hand, if this condition evaluates as false (there is not a minimum battery level to execute the function), so the options menu can be presented (*showConMenu()* is executed).

## 4.3 Concluding Remarks

In this chapter we applied the mapping process presented in Chapter 3 in two case studies: Pulse Oximeter and ConnectOK.

The Pulse Oximeter is an equipment used inside Critical Care Units for measuring the blood oxygen saturation of patients.

The ConnectOK is a mobile device used to reading water, energy and gas consumption.

We applied the mapping process in these two case studies in order to analyse and verify some interesting properties of that systems. First, we obtained the CPN model and then open this model in CPN Tools to analyse and verify some properties. At the analysis phase we investigate some common properties, such as liveness, boudedness, repetitiveness, fairness and conservativeness. After analysing this properties, we constructed some specific queries to verify some properties. These queries are built based on model checking language (CPN-ML) and State Space Queries. These queries analyse the state space to verify if a condition is satisfied.

# Chapter 5

# Conclusion

*This chapter brings a summary of the presented work and mentions some future works.*

Nowadays, embedded systems are present in almost any human interacting environment and activities. The crescent adoption of embedded-system-controlled machines is direct related to the decreasing costs of such systems.

Due to the cost and the complexity of embedded systems, containing multiple hardware and software components, sophisticated communication structure, the variety of possible solutions, performance, energy consumption constraints, correctness and robustness, it is essential high-level system design tools and methods, where functional and architectural description validation and verification might be carried out. Improving system reliability can be carried out by simulation or through formal analysis/verification that is quite attractive because they spare exhausting simulations.

Over the last years, scenario based mechanisms have been adopted as an interesting alternative for specifying system's requirements. A more recent way to specify requirements, which is popular in the realm of object-oriented systems, is the adoption of Message Sequence Charts (MSCs). However, MSCs have some drawbacks, since it can not specify what must occur for all system executions, as well as it is unable of specifying anti-scenarios.

The LSC language reduces some shortcomings inherent to MSC based models, such as allowing the possibility of specifying liveness and anti-scenarios. LSC allows modelers distinguishing between possible and necessary behavior both globally (existential and universal charts), on the level of an entire chart, and locally, when specifying events, conditions, and progress over time within a chart. The *Play-Engine* is a powerful tool, in which LSC scenarios can be described and simulated. However, it does not allow an analysis and verification of system's properties.

Petri Net (PN) is, nowadays, a general tool for specifying a family of formal specification models suit for representing synchronization, concurrency or resource sharing. Therefore, PN could be used as a possible approach to analysis and verification of system's properties.

This work presented how to mapping the Live Sequence Chart (LSC) language into an equivalent Coloured Petri Net (CPN) model [33, 34], in which the obtained CPN model could be analysed and verified. As LSC language has data-types and adopts high-level concepts such as method invocation, Coloured Petri Nets have been adopted as a suit Petri net variant since it supports complex data-types and a programming language (CPN-ML) that improves value's handling. Therefore, the proposition of a CPN model for LSC allows verification and analysis of systems described in LSC, hence, contributing for increasing designers' confidence on the system development process and reducing risks that may lead to project failure.

Throughout Chapter 3, the steps that must be applied in order to obtain the corresponding CPN model (for each individual construction) have been described. At the end of Chapter 3, we compared LSC and CPN semantics by considering an example, which contains some of the presented LSC constructions. This comparison method does not aim, at this point of our research, being complete or even validating the mapping method.

After describing the steps to obtain a corresponding CPN model for an LSC chart, we applied the proposed methodology in two case studies, presented in Chapter 4. In these case studies, the CPN models were obtained through LSC2CPN engine, which automates the LSC translation. After translating the LSC specifications, the obtained CPN models are analysed, in which some interesting properties are discussed. Besides analysis, some specific properties were verified using ASKCTL formulas as a model checking language and using state space queries.

This work brings an important contribution since it describes how to mapping Live Sequence Chart into Coloured Petri Nets for properties analysis and verification of specifications based on LSC language, which may be useful in the early stages of an embedded system project, reducing risks that may lead to a project failure. However, there are some enhancements that should be considered.

The proposed methodology does not map the whole set of LSC constructions, classes, symbolic instances and forbidden elements are not discussed. Classes and symbolic instances permit to specify more complicated and more powerful scenarios. Through forbidden elements it is possible to specify a more direct and flexible means for anti-scenarios.

Another drawback takes respect to the comparison process for LSC and CPN semantics, presented in Chapter 3, that should be applied for more examples, however, the presented example contains a representative set of the LSC constructions explored in this work.

The verification process adopted in the case studies can be improved by supplying an interface, in which the user could construct the queries without the need to know the model checking language syntactic. The user could define some parameters, then submits the verification request and waits for the result, that could be presented as more detailed and explained message.

In order to validate the correctness of the proposed methodology, a more general validation process should be considered. A process algebra semantics for the LSC language may help in the establishment of this validation process.

Additional work is necessary on the LSC2CPN engine for improving the automation of the mapping process. Actually, the engine supplies the CPN model as a CPN Tool [1] format. So the compatibility could be improved to supply the CPN model as a more general format in order to permit that the CPN model can be treated in different tools.

Embedded software has become much harder to design due to the diversity of requirements and high complexity. In such systems, correctness and timeliness verification is an issue to be concerned. If we ally to this mapping process, the capability to synthesize code to automatically generates a "safe" program source code [35], the risks of an embedded project could be reduced.

Nowadays, time constraints can not be attached to individual LSC events. Once an enhanced version of LSC language with support of time constraints for events, the presented methodology could be improved to deal with these time constraints and provide a way to execute a performance evaluation of an embedded system described using this enhanced LSC language.

# Appendix A

# Support Engine

This appendix presents the LSC2CPN support engine, which is applied to automatize the process of CPN model generation from LSC inscriptions.

The goal of this support engine is not carry through analysis and verification of properties of the specified system. This engine applies the presented mapping methodology, in which, a CPN model is obtained for each LSC construction. Later, this individuals CPN models are joined in order to provide a final CPN model, which can be analysed and verified with the aid of a tool that offers such support, in this case the CPN Tools [1].

The LSC2CPN engine was built using JAVA [3] technology together with JDOM [4], which is an additional library that allows manipulate XML [2] files in an easy way. The input to this mapping engine is a XML file generated by Play-Engine, which contains the whole system specification, modeled through LSC scenarios. After processing this input file, the LSC2CPN engine applies the mapping process and generates a XML file as an output, which contains the final CPN model that can be loaded into a specific tool (CPN Tools) in order to execute properties' analysis and verifying of the modeled system.

The LSC2CPN operation flow can be summarized as follows:

1. Load the XML file provided by Play-Engine;

2. Create the corresponding CPN structure to represent LSC types and classes defined by the user;

3. Create CPN variables to represent each instance inside each LSC chart available in the specification;

4. Obtain the individual CPN model for each LSC construction inside a LSC chart;

5. Join the CPN models obtained in the previous step, in order to find a CPN model that represents a LSC chart;

6. Join all CPN models that represents an LSC chart;

7. Generate a XML file with the final CPN model that represents the LSC specification passed as an input.

This support engine assists the process of properties' analysis and verification of the modeled system, but does not automate the whole process since each system has its particularities, becoming essential the availability of a professional who knows CPN in order to realize the custom verification of the modeled system.

# Appendix B

# Basic Theory

In this appendix it is presented the basic concepts on logic, theory of sets and functions. Many of the definitions presented here can be found in [47, 50].

## B.1 Logic

A sentence (or proposition) is an expression which is either true or false. The sentence “2 + 2 = 4” is true, while the sentence “Π is rational” is false. It is, however, not the task of logic to decide whether any particular sentence is true or false. In fact, there are many sentences whose truth or falsity nobody has yet managed to establish; for example, the famous Goldbach conjecture that “every even number greater than 2 is a sum of two primes”.

Since there are expressions which are sentences under our definition, we proceed to discuss ways of connecting sentences to form new sentences.

Let $p$ and $q$ denote sentences.

**Definition.** (Conjunctions) We say that the sentence $p \wedge q$ ($p$ and $q$) is true if the two sentences $p$, $q$ are both true, and is false otherwise.

**Definition.** (Disjunction) We say that the sentence $p \vee q$ ($p$ or $q$) is true if at least one of two sentences $p$, $q$ is true, and is false otherwise.

**Remark.** To prove that a sentence $p \vee q$ is true, we may assume that the sentence $p$ is false and use this to deduce that the sentence $q$ is true in this case. For if the sentence $p$ is true, our argument is already complete, never mind the truth or falsity of the sentence $q$.

**Definition.** (Negation) We say that the sentence $\overline{p}$ (not $p$) is true if the sentence $p$ is false, and is false if the sentence $p$ is true.

**Definition.** (Conditional) We say that the sentence $p \rightarrow q$ (if $p$, then $q$)

is true if the sentence $p$ is false or if the sentence $q$ is true or both, and is false otherwise.

**Remark.** It is convenient to realize that the sentence $p \rightarrow q$ is false precisely when the sentence $p$ is true and the sentence $q$ is false. To understand this, note that if we draw a false conclusion from a true assumption, then our argument must be faulty. On the other hand, if our assumption is false or if our conclusion is true, then our argument may still be acceptable.

**Definition.** (Double Conditional) We say that the sentence $p \leftrightarrow q$ ($p$ if and only if $q$) is true if the two sentences $p$, $q$ are both true or both false, and is false otherwise.

## B.2 Functions

Let $A$ and $B$ be sets. A function (or mapping) $f$ from $A$ to $B$ assigns to each $x \in A$ an element $f(x)$ in $B$. We write $f : A \rightarrow B : x \mapsto f(x)$ or simply $f : A \rightarrow B$. $A$ is called the domain of $f$, and $B$ is called the co-domain of $f$. The element $f(x)$ is called the image of $x$ under $f$. Furthermore, the set $f(B) = \{y \in B : y = f(x) \text{ for some } x \in A\}$ is called the range or image of $f$.

Two functions $f : A \rightarrow .B$ and $g : A \rightarrow .B$ are said to be equal, denoted by $f = g$, if $f(x) = g(x)$ for every $x \in A$.

It is sometimes convenient to express a function by its graph $G$. This is defined by

$$G = (x, f(x)) : x \in A = (x, y) : x \in Aandy = f(x) \in B.$$

**Definition.** We say that a function $f : A \rightarrow B$ is one-to-one if $x1 = x2$ whenever $f(x1) = f(x2)$.

**Definition.** We say that a function $f : A \rightarrow B$ is onto if for every $y \in B$, there exists $x \in A$ such that $f(x) = y$.

**Remarks.** If a function $f : A \rightarrow B$ is one-to-one and onto, then an inverse function exists. To see this, take any $y \in B$. Since the function $f : A \rightarrow B$ is onto, it follows that there exists $x \in A$ such that $f(x) = y$. Suppose now that $z \in A$ satisfies $f(z) = y$. Then since the function $f : A \rightarrow B$ is one-to-one, it follows that we must have $z = x$. In other words, there is precisely one $x \in A$ such that $f(x) = y$. We can therefore define an inverse function $f^{-1} : B \rightarrow A$ by writing $f^{-1}(y) = x$, where $x \in A$ is the unique solution of $f(x) = y$.

**Remarks.** Consider a function $f : A \rightarrow B$. Then $f$ is onto if and only if for every $y \in B$, there is at least one $x \in A$ such that $f(x) = y$. On the other hand, $f$ is one-to-one if and only if for every $y \in B$, there is at most one $x \in A$ such that $f(x) = y$.

Suppose that $A$, $B$ and $C$ are sets and that $f : A \to B$ and $g : B \to C$ are functions. We define the composition function $g \circ f : A \to C$ by writing $(g \circ f)(x) = g(f(x))$ for every $x \in A$.

**Associative law.** Suppose that $A$, $B$, $C$ and $D$ are sets, and that $f : A \to B, g : B \to C$ and $h : C \to D$ are functions. Then $h \circ (g \circ f) = (h \circ g) \circ f$.

## B.3 Sets

A set is usually described in one of the two following ways:

- By enumeration, e.g. 1, 2, 3 denotes the set consisting of the numbers 1, 2, 3 and nothing else;

- By a defining property (sentential function) $p(x)$. Here it is important to define a universe $U$ to which all the $x$ have to belong. We then write $P = \{x : x \in U$ and $p(x) is true\}$ or, simply, $P = \{x : p(x)\}$.

Suppose that the sentential functions $p(x)$ and $q(x)$ are related to sets $P$, $Q$ with respect to a given universe, i.e. $P = \{x : p(x)\}$ and $Q = \{x : q(x)\}$. It is defined:

- The intersection $P \cap Q = \{x : p(x) \wedge q(x)\}$;

- The union $P \cup Q = \{x : p(x) \vee q(x)\}$;

- The complement $\overline{P} = \{x : \overline{p(x)}\}$;

- The difference $P/Q = \{x : p(x) \wedge \overline{q(x)}\}$.

The above are also sets. It is not difficult to see that:

- $P \cap Q = \{x : x \in P$ and $x \in Q\}$;

- $P \cup Q = \{x : x \in P$ or $x \in Q\}$;

- $\overline{P} = \{x : x \notin P\}$;

- $P/Q = \{x : x \in P$ and $x \notin Q\}$.

The set $P$ is a subset of set $Q$, denoted by $P \subseteq Q$ or by $Q \supseteq P$, if every element of $P$ is an element of $Q$. In other words, if we have $P = \{x : p(x)\}$ and $Q = \{x : q(x)\}$ with respect to some universe $U$, then we have $P \subseteq Q$ if and only if the sentence $p(x) \rightarrow q(x)$ is true for all $x \in U$.

Two sets $P$ and $Q$ are equal, denoted by $P = Q$, if they contain the same elements, i.e. if each is a subset of the other, i.e. if $P \subseteq Q$ and $Q \subseteq P$.

Furthermore, $P$ is a proper subset of $Q$, denoted by $P \subset Q$ or by $Q \supset P$, if $P \subseteq Q$ and $P \neq Q$.

The following results on set functions can be deduced from their analogues in logic.

**Distributive Law.** If $P$, $Q$, $R$ are sets, then:

(a) $P \cap (Q \cup R) = (P \cap Q) \cup (P \cap R)$;

(b) $P \cup (Q \cap R) = (P \cup Q) \cap (P \cup R)$.

**De Morgan Law.** If $P$,$Q$ are sets, then with respect to a universe $U$:

(a) $\overline{(P \cap Q)} = \overline{P} \cup \overline{Q}$;

(b) $\overline{(P \cup Q)} = \overline{P} \cap \overline{Q}$.

In general, consider a sentential function of the form $p(x)$, where the variable $x$ lies in some clearly stated set. It can be consider the following two sentences:

- $\forall x, p(x)$ (for all $x$, $p(x)$ is true);
- $\exists x, p(x)$ (for some $x$, $p(x)$ is true).

**Definition.** The symbols $\forall$ (for all) and $\exists$ (for some) are called the universal quantifier and the existential quantifier respectively.

Note that the variable $x$ is a "dummy variable". There is no difference between writing $\forall x, p(x)$ or writing $\forall y, p(y)$.

# Appendix C

# Glossary

**FIFO –** First in, first out. An asset-management and valuation method in which the assets produced or acquired first are sold, used or disposed of first.

**Java –** A programming language introduced by Sun Microsystems. Java is a multi-platform, platform-independent, object oriented programming language. Java programs are not compiled, but rather interpreted as run.

**GSM –** Global System for Mobile communications, the most widely used digital mobile phone system and the de facto wireless telephone standard in Europe. Originally defined as a pan-European open standard for a digital cellular telephone network to support voice, data, text messaging and cross-border roaming. GSM is now one of the world's main 2G digital wireless standards.

**GPRS –** A packet switching technology for GSM networks. It's an advanced data transmission mode that does not require a continuous connection to the Internet, as with a standard home modem. Instead, GPRS uses the network only when there is data to be sent, which is more efficient.

**ITU –** International Telecommunication Union, an intergovernmental organization through which public and private organizations develop telecommunications. The ITU was founded in 1865 and became a United Nations agency in 1947. It is responsible for adopting international treaties, regulations and standards governing telecommunications. The standardization functions were formerly performed by a group within the ITU called CCITT, but after a 1992 reorganization the CCITT no longer exists as a separate entity.

# Bibliography

[1] *CPN Tools*. http://wiki.daimi.au.dk/cpntools/ cpntools.wiki.

[2] *Extensible Markup Language*. http://www.xml.org.

[3] *Java Technology*. http://java.sun.com.

[4] *JDOM API*. http://www.jdom.org.

[5] *Products Web Page*. http://www.ilogix.com/fs_prod.htm.

[6] *Rational RealTime tool*. http://www.rational.com.

[7] G. Gullekson B. Selic and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.

[8] Luciano Baresi and Mauro Pezzè. *Improving UML with Petri Nets*, volume 44. 2001.

[9] T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS*, volume 14, pp. 25-59. Computer Network and ISDN Systems, 1987.

[10] Y. Bontemps and P. Heymans. *Turning High-Level Live Sequence Charts into Automata*. 2002.

[11] G. Booch. *Object-Oriented Analysis and Design, with Applications*. Benjamin-Cummings, San Mateo, California, 1994.

[12] A. Bunker and K. Slind. *Property Generation for Live Sequence Charts*. Technical report, University of Utah, 2003.

[13] A. Burns and A. Willings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.

[14] CCITT. *CCITT Recommendation Z.120: Message Sequence Chart (MSC)*. Geneva, 1992.

[15] P. Chen. *The Entity-Relationship Model: Toward a Unified View of Data*, volume 1. 1976.

[16] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice Hall, Upper Saddle River, Nova Jersey, 1994.

[17] W. Damm and D. Harel. *LSCs: Breathing Life into Message Sequence Charts*. Kluwer Academic, 2001.

[18] J. Rumbaugh G. Booch and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[19] HJ Genrich and K. Lautenbach. *System modeling with high level Petri Nets*, volume 13. 1981.

[20] A. Pnueli Y. Lu H. Kugler, D. Harel and Y. Bontemps. *Temporal Logic for Scenario-Based Specifications*. Lectures Notes in Computer Science. Springer-Verlag, 2005.

[21] D. Harel. *Biting the Silver Bullet: Toward a Brighter Future for System Development*. 1992.

[22] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*, volume 8. The Weizmann Institute of Science, Israel, 1984.

[23] D. Harel and E. Gery. *Executable Object Modeling with Statecharts*. 1997.

[24] D. Harel and R. Marelly. *Come, Lets Play: Scenario-Based Programming Using LSCs and Play-Engine*. Springer-Verlag, 2003.

[25] C. A. R. Hoare. *Communicating Sequential Processes*, volume 21. 1978.

[26] J. Campos J. Merseguer and E. Mena. *Performance Evaluation for the Design of Agent-based Systems: A Petri Net Approach*. 1998.

[27] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press/Addison-Wesley, 1992.

[28] K. Jensen. *An Introduction to the Practical Use of Coloured Petri Nets*, volume 1492. Springer-Verlag, 1993.

[29] K. Jensen. *An Introduction to the Theoretical Aspects of Coloured Petri Nets*, volume 803. Springer-Verlag, June 1993.

[30] M. N. Oliveira Junior. *Desenvolvimento de Um Protótipo para a Medida Não Invasiva da Saturação Arterial de Oxigênio em Humanos – Oxímetro de Pulso*. MSc Thesis, UFPE, August, 1998.

[31] J. Klose and H. Wittke. *An Automata Based Interpretation of Live Sequence Chart*. In Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), 2001.

[32] O. Kluge. *Compositional Semantics for Message Sequence Charts based on Petri Nets*. Elektrotechnik und Informatik, September, 2002.

[33] M. Oliveira R. Barreto L. Amorim, P. Maciel and E. Tavares. *A Methodology for Mapping Live Sequence Chart to Coloured Petri Net*. IEEE, 2005.

[34] M. Oliveira R. Barreto L. Amorim, P. Maciel and E. Tavares. *Mapping Live Sequence Chart to Coloured Petri Nets for Analysis and Verification of Embedded Systems*. ACM, 2006.

[35] P. Maciel E. Tavares M. Oliveira A. Bessa L. Amorim, R. Barreto and R. Lima. *A Methodology for Software Synthesis of Embedded Real-Time Systems Based on TPN and LSC*. Springer-Verlag, December 2005.

[36] D. Lyonnard. *Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip*. New Orleans, 2001.

[37] G. De Micheli and L. Benini. *Networks-on-Chip: A New Paradigm for Systems-on-Chip Design*. Paris, 2002.

[38] R. Milner. *Calculus Communicating Systems*. Springer-Verlag, 1980.

[39] T. Murata. *Petri Nets: Properties, Analysis and Applications*, volume 77(4). IEEE, April 1989.

[40] OMG. *Documentation of the Unified Modeling Language*. http://www.omg.org.

[41] R. Lins P. Maciel and P. Cunha. *Introdução às Redes de Petri e Aplicações*. X Escola de Computação Campinas-SP, 1996.

[42] B. Selic P. Ward and G. Gullekson. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[43] D. Park. *Concurrency and Automata on Infinite Sequences*, volume 104. Springer-Verlag, March 1981.

[44] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[45] J. Peterson. *Petri Net Theory and The Modeling of Systems*. Englewood Cliffs, Pretice-Hall, 1981.

[46] C. Petri. *Fundamentals of a Theory of Asynchronous Information Flow*. 1962.

[47] Iain Phillips. *Discrete Mathematics and Algorithm Analysis*. Lecture Notes for Computer Science.

[48] W. Reisig. *Petri Nets, An Introduction*. Springer-Verlag, 1985.

[49] W. Reisig and G. Rozenberg. *Lectures on Petri Nets I: Advances in Petri Nets, Lecture Notes in Computer Science 1491*. Springer-Verlag, 1998.

[50] K. H. Rosen. *Discrete Mathematics and its Applications*. Prentice Hall, second edition, 1999.

[51] J. Rumbaugh. *Object-Oriented Modeling and Design*. Prentice Hall, Upper Saddle River, Nova Jersey, 1991.

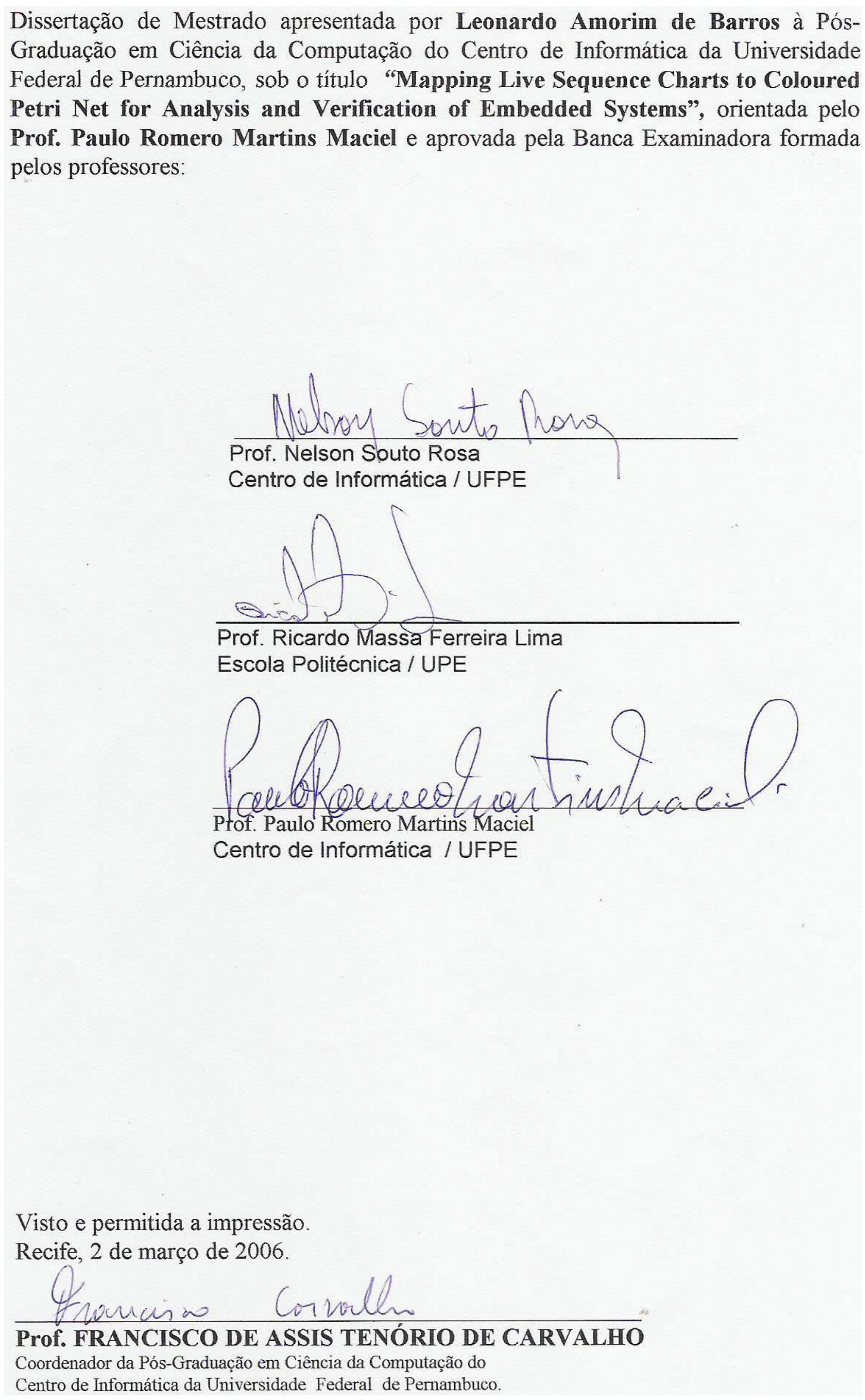
[52] J. Sun and J. Dong. *Model Checking Live Sequence Charts*. Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005.

[53] R. Paul C. Fan W. T. Tsai, L. Yu and X. Liu. *Rapid Scenario-Based Simulation and Model Checking for Embedded System*. Proceedings of International Conference on SEA, 2003.

[54] ISO/IEC JTC1/SC21 WG7. *Enhancements to LOTOS*. September, 2001.

[55] P. Heymans Y. Bontemps and Pierre-Yves Schobbens. *From Live Sequence Charts to State Machines and Back: A Guided Tour*, volume 31. December, 2005.

[56] R. Zurawski and M. Zhou. *Petri Nets and Industrial Applications: A Tutorial*, volume 41. 1994.



Dissertação de Mestrado apresentada por **Leonardo Amorim de Barros** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Mapping Live Sequence Charts to Coloured Petri Net for Analysis and Verification of Embedded Systems"**, orientada pelo **Prof. Paulo Romero Martins Maciel** e aprovada pela Banca Examinadora formada pelos professores:

[illegible]

Prof. Nelson Souto Rosa
Centro de Informática / UFPE

[illegible]

Prof. Ricardo Massa Ferreira Lima
Escola Politécnica / UPE

[illegible]

Prof. Paulo Romero Martins Maciel
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 2 de março de 2006.

[illegible]

**Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO**
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Figure C.1: Aprovação