Colored Petri Nets in the Animation of UML Models for Requirements Validation

Sérgio António Real de Oliveira

Dissertação submetida à Universidade do Minho
para obtenção do grau de Mestre em Sistemas de Informação,
elaborada sob a orientação científica dos Professores:

Ricardo Jorge Silvério de Magalhães Machado

e

Guilherme Augusto Borges Pereira

Departamento de Sistemas de Informação
Escola de Engenharia
Universidade do Minho
Guimarães, Julho de 2006

ESCOLA DE ENGENHARIA DA UNIVERSIDADE DO MINHO
DEPARTAMENTO DE SISTEMAS DE INFORMAÇÃO

SEMAG - SOFTWARE ENGINEERING AND MANAGEMENT RESEARCH GROUP
CENTRO DE INVESTIGAÇÃO ALGORITMI

Colored Petri Nets in the Animation of UML Models for Requirements Validation

Sérgio António Real de Oliveira

Licenciado em Engenharia Sistemas e Informática pela
ESCOLA DE ENGENHARIA DA UNIVERSIDADE DO MINHO, 1993

Dissertação submetida à Universidade do Minho
para obtenção do Grau Académico de Mestre em Sistemas de Informação,
elaborada sob a orientação científica dos Professores:

Ricardo Jorge Silvério de Magalhães Machado
Professor Auxiliar do Departamento de Sistemas de Informação
da ESCOLA DE ENGENHARIA DA UNIVERSIDADE DO MINHO

e

Guilherme Augusto Borges Pereira
Professor Auxiliar do Departamento de Produção e Sistemas
da ESCOLA DE ENGENHARIA DA UNIVERSIDADE DO MINHO

Guimarães, Julho de 2006

Acknowledgements

After many years working outside the academic world, having become one of the many victims of the "collateral effects" of the so-called "country-on-its-knees"[1] economical situation, was all that I needed for deciding to give a chance to an ever seducing idea that a steady job managed to keep in a lethargic state for too long: to try-out my capabilities for the work as a researcher. Having been pushed by those circumstances, and being rather out of the usual age for such an experiment, this was not only a demanding task in what concerns to perseverance, but also threatening to my self-confidence in the eventuality of failure. However, the more than expectable support I received from various persons helped me to overcome all those fears. I am very grateful to all those persons, some of which I must in particular refer.

The first persons to whom I want to express my gratitude are my supervisors – Prof. Ricardo Machado and Prof. Guilherme Pereira – for their constant availability, for their wise orientation in maintaining my work within high scientific standards, for preventing me from deriving to unrealistic goals, and for their flexibility to accept the adaptation of those goals to my personal preferences, all that done in the most friendly cordiality. I must express a particular reference to Prof. Ricardo Machado, for his constant encouragement in many ways, as was the case of conceding me the privilege of choosing me to present our paper, the first one in which I participated, in the CPN05 Workshop, at the University of Aarhus, Denmark.

For their determinant contribution to the development of the uPAIN animation prototype, I must thank Marco Couto and Patrícia Pinto, also co-authors of the paper presented in the CPN05 Workshop, with whom I have shared my working hours, as lab mates, in a most friendly good disposition.

[1] A free translation for the Portuguese expression "país de tanga".

I also thank Kristian Bisgaard Lassen, of the Department of Computer Science of the University of Aarhus, Denmark, for sharing with us know-how and techniques on the use of CPN-Tools and the BRITNEY animation tool, without which the uPAIN animation prototype would not have reached the quality that it has. My gratitude must also be expressed to Prof. Kurt Jensen, Prof. Søren Christensen and Prof. Jens Bæk Jørgensen of the Department of Computer Science of the University of Aarhus, whose interest in our application of those techniques not only made possible the participation of Kristian, but also functioned as an important stimulus.

I thank Prof. João Miguel Fernandes, from the Department of Informatics of the Engineering School of the University of Minho, for one opportune strategic advice he gave me, and for his good will in conceding me free access to his lab.

I also want to thank Prof. João Álvaro Carvalho and Prof. Luís Amaral, respectively former and present directors of the Department of Information Systems of the Engineering School of the University of Minho, for their determinant engagement in providing a particular environment that stimulates a special spirit for the research work at the department. Another expression of my recognition goes to the director of the MSc course in Information Systems, Prof. Manuel Filipe Santos, for his commitment in giving always the best support to the MSc students.

Special thanks to my wife Elisabete, my daughter Catarina and my son Miguel for their understanding and love, and for being so fantastic. Special thanks also to Zé Pedro, Arlindo and Zé Luís, for their friendship, enthusiastic support, and the indispensable beer meetings. They all provided me the strength to overcome all weaknesses.

Last, but not least, I thank my colleagues Miguel Ferreira, Cesar Ávila, Óscar Ribeiro and Paula Monteiro, for their interest, and their availability to discuss ideas and helping in several ways.

Abstract

Functional requirements must be validated by the client of the software project (managers, users and stakeholders in general). However, requirements validation is a critical task in any engineering project. Presenting static requirements models to the stakeholders is not sufficient, since it does not allow stakeholders, who do not have computer science background, to discover all the interdependencies between the elicited requirements. With the mere presentation and explanation of UML (*Unified Modelling Language*) requirements models, even the most simple and intuitive ones, it is not easy for the software engineer to be confident on the stakeholders' requirements validation. This thesis describes an approach, based on the construction of executable interactive animation prototypes, to support the validation of functional requirements, where the system to be built must explicitly support the interaction among people within a pervasive cooperative workflow execution. A demonstration case from a real project is used to illustrate the proposed approach.

Redes de Petri Coloridas na Animação de Modelos UML para Validação de Requisitos

Resumo

Os requisitos funcionais de software têm de ser validados pelo cliente do projecto de desenvolvimento (gestores, utilizadores e todas as partes interessadas - *stakeholders* - em geral). Mas a validação de requisitos é uma tarefa crítica em qualquer projecto de engenharia. A mera apresentação de modelos estáticos aos *stakeholders* não é suficiente, porque os *stakeholders* sem formação em ciências da computação não são capazes de descobrir todas as interdependências entre os requisitos, que estão implícitas nesses modelos. Recorrendo apenas à apresentação e explicação de modelos de requisitos em UML (*Unified Modelling Language*), mesmo os mais simples e intuitivos, não é fácil ao engenheiro de software sentir confiança na validação dos requisitos pelos *stakeholders*. Esta dissertação descreve uma abordagem, baseada na construção de protótipos executáveis de animação interactiva, para suportar a validação de requisitos funcionais de um sistema, a construir, que suporte explicitamente a interacção entre pessoas na execução de um processo cooperativo *pervasive* (intrusivo). Um caso de demonstração de um projecto real é usado para ilustrar a abordagem proposta.

To my mother Maria
my wife Elisabete
my daughter Catarina
and my son Miguel

Table of Contents

Index of Figures

Chapter 1

Introduction

Clients of software projects (users and stakeholders in general) and developers (system designers and requirements engineers) have, naturally, different points of view towards requirements. Actually, a requirement can be defined as "something that a client needs" but also, from the point of view of the system designer or the requirements engineer, as "something that must be designed". The IEEE 610 standard [IEEE, 1990] defines a requirement as: (1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system, or system component, to satisfy a contract, standard, specification, or other formally imposed documents; (3) a documented representation of a condition or capability as in (1) or (2).

Taking into account these two distinct perspectives, two different categories for requirements can be conceived:

- *User requirements* result directly from the requirements elicitation task [Zowghi and Coulin, 2005], in an effort to understand the stakeholders' needs. They are typically described in natural language and with informal diagrams, at a relatively low level of detail. User requirements are focused in the problem domain and are the main communication medium between the stakeholders and the developers, at the analysis phase.
- *System requirements* result from the developers' effort to organize the user requirements at the solution domain. They, typically, comprise abstract models of the system [Machado *et al.*, 2005a], at a relatively high level of detail, and con-

stitute the first system representation to be used at the beginning of the design phase.

The correct derivation of system requirements from user requirements is an important objective, because it assures that the design phase is effectively based on the stakeholders' needs. Some existent techniques [Liang, 2003; Whittle *et al.*, 2005; Krüger *et al.*, 1999; Machado et al, 2005 b] can be used to support the transformation of user requirements models into system requirements models, by manipulating the corresponding specifications. This also guarantees that no misjudgment is arbitrarily introduced by the developers during the process of system requirements specification.

However, this effort for maintaining the continuity of the models' functional coherence, by applying controlled transformational techniques, can be worthless if the user requirements models are not effectively validated. Typically, the confrontation of stakeholders with static requirements models is not enough, since stakeholders without computer science education are not able to discern all the interdependencies between the elicited requirements. With the mere presentation and explanation of UML2 (Unified Modeling Language) requirements models (use case diagrams and some kind of sequence diagrams), even the most simple and intuitive ones, it is not easy for the software engineer to be confident on the stakeholders' requirements validation.

1.1 Workflow perspective

The production of goods or services requires the execution of sets of operations that must be performed according to predefined processes. Simple processes, like for instance the making of handicraft pieces, can be executed by a single person. But the majority of processes that run in today's organizations are typically executed by several participants – people and/or machines (including computers and their software). The reason behind this new reality lies on the fact that most of the goods or services produced nowadays have such a complexity, that they require several high-level skills and competencies, that makes them impossible to be executed by a single participant (person or machine), because the number of different high-level competencies that a participant is capable of possessing is always limited. Consequently, each process must be subdivided into sub-processes and activities, in a number that, at maximum, equals the

[2] http://www.uml.org/

number of specific competencies required for the process as a whole, in order that each participant may be assigned only the process's activities for which the participant has the required competencies.

In most processes, there are some activities that may be executed simultaneously, but when there are rules that impose precedence between activities, these must necessarily be executed sequentially (a situation that also occurs, at least partially, in almost every process). Therefore, the execution of a business process may be seen as the flow of work (represented by documents, information, materials or tasks) passing from one participant to another, according to rules that are specific to the process. It is this notion of controlled flow of work that originated the designation **workflow**.

The Workflow Management Coalition (WfMC) defines business process and workflow as [WFMC, 1999]:

> *"A business process is a set of one or more linked procedures, which collectively realize a business objective or policy goal, normally within the context of an organizational structure, defining functional roles and relationships".*
>
> *"A workflow is the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules."*

From what has been stated previously, and this definition of workflow, the difference is subtle, residing mainly in the addition of the term *automation*. Yet, this subtle addition carries within it a consequence of major importance. It enforces the use of information technology (IT) (e.g. a workflow management system - WMS) to control the whole process, which, in turn, means that it implies the need of, among other things, modeling the process in a representation that is adequate for IT specialists to convert it into a process definition of a WMS.

Depending on the characteristics inherent to the nature of each business process, four types of workflows are widely considered, including by the WfMC:

> **Production Workflow –** It corresponds to the automation of processes that are made of a large number of similar and highly repetitive tasks, aiming at productivity maximization through the automation of almost all of

the tasks, reducing human intervention to the resolution of exception situations. Modeling these workflows is a heavy task and generates complex and rigid models, requiring a high consumption of time, both in the initial definition and subsequent changes (which, for that reason, are expected to be rare).

Administrative Workflow – It is characterized by the tendency to involve a large number of people, and by being applied to processes that require frequent changes. For this reason, it is desirable that they are kept easy to model, and that the model be as flexible as possible, even at the cost of a less optimized productivity.

Collaborative workflow – It applies to situations where several teams work together for the realization of a common goal. It can be the case of project oriented small groups, or highly dispersed geographically people sharing common interests. With the globalization phenomenon and the ever increasing presence of the INTERNET in the strategy of the organizations, this kind of workflow assumes a crucial role in the coordination of processes, which typically involve the cooperation among several organizations around the same goals, as is the case of e-business. The flexibility of quick adaptation to new products or services and the coordination are its main requirements, and productivity is treated as a second priority.

***Ad-Hoc* workflows** – These workflows are adequate to processes which are subject to changes to the initial definition at any moment. The priority is on the haste of the process of defining the workflows and on how easy it is to change process rules to specific cases, originating multiple variants of the process.

The characterization of these four types of workflow shows how different workflows can be in terms of several properties. Yet, it is obvious that, although there is an important diversity of factors that influence processes, the main criterion for the evaluation of a process is its functional correctness, which is determined by the process definition. This is why modeling is considered a key issue by workflow researchers, who have been investing a great deal of effort in creating artifacts and improving their capability

of supporting the whole diversity of both functional and non-functional requirements of business processes.

In [Aalst *et al.*, 2003] three tendencies of the evolution of information technology (IT) are identified: (1) an increase in functionality, translated into an increase of the number of software layers in computer systems; (2) an increased focus on processes and the corresponding decrease of the relative importance of the data perspective; (3) a progressive increase of the practice of redesign and organic growth, as a response to the constant need to the more and more frequent need for organizations to adapt their business processes to constant market changes. These tendencies explain, not only the growing interest on Business Process Management (BPM) [BPMG] and WMSs [3] (like Staffware [Staffware], MQSeries [MQSeries] and COSA [COSA], to name just a few) and case handling systems (like FLOWer) [FLOWer], but also on process modeling.

The workflow modeling languages of the commercially available workflow management systems are based on one of the two groups of languages that are used by the scientific community studying workflows: (1) Process Algebras and (2) Petri nets. The history of the application of both groups of languages to the workflow area has approximately the same age. Although the genesis of Petri nets ascends to 1962, with the PhD dissertation of Carl Adam Petri [Petri, 1962], according to [Aalst *et al.*, 2003] the use of variants of Petri nets to model office proceedings (in what can be considered the precursors of the WMSs – the so called "Office Information Systems") took place only in the seventies, with the initiative of researchers like Clarence (Skip) Ellis [Ellis, 1979], Anatol Holt [Holt, 1985] and Michael Zisman [Zisman, 1977]. According to [Best and Koutny, 2004], the first ideas of process algebras were introduced in the same age by Tony Hoare [Hoare, 1978] and Robin Milner [Milner, 1980]. As also mentioned by [Best and Koutny, 2004], the relations between both groups of languages have been studied during their development. Possibly motivated by commercial interests, some apologists of one, or the other, group of languages have incurred in exaggerated appreciations of the qualities of their preferred languages, as commented in an unpublished paper [Aalst, 2003], about the relative strengths and weaknesses of process algebras and Petri nets, with a clear conclusion in favor of Petri nets.

[3] An extensive list can be found in http://www.doconsite.co.uk/DirectoryPages/Systems/wmsbysupp.cfm.

1.2 SWEBOK perspective

Being a subject of major importance in software engineering, the Guide to Software Engineering Body of Knowledge (SWEBOK) - a project of the IEEE Computer Society Professional Practices Committee - treats requirements as the first *Knowledge Area* (KA) of software engineering, entitled precisely *Software Requirements*. In the topic concerning the definition of software requirement, it is possible to read in the SWEBOK [IEEE, 2004, pp. 2-1]:

> *"A software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem. The problem may be to automate part of a task of someone who will use the software, to support the business processes of the organization that has commissioned the software, to correct shortcomings of existing software, to control a device, and many more. The functioning of users, business processes, and devices is typically complex. By extension, therefore, the requirements on particular software are typically a complex combination of requirements from different people at different levels of an organization and from the environment in which the software will operate."*

There is nothing in this excerpt from the SWEBOK contradicting the definition given by the IEEE 610 standard. It is just introducing a motivation for the need for documenting, modeling and validating requirements. In fact, the SWEBOK presents several classifications of requirements (*product and process requirements*, *functional and non-functional requirements*, *emergent properties*, *quantifiable requirements*, *system requirements*) according to different perspectives, whose descriptions allow summarizing them into the classification given by the definition of the IEEE 610 standard.

In the SWEBOK, modeling is treated in the subtopic 4.2 – *Conceptual Modeling* – of the 4th topic – *Requirements Analysis* – of the Software Requirements KA. It is considered there that the purpose of conceptual modeling is more helping to understand the problem, rather than initiating the design of the solution. Conceptual modeling is thus associated to the treatment of the user requirements, in the sense that before proceeding to the next modeling phase – the modeling of the solution – their validation, by the users, is required, as a means of certifying that they were fully understood by the software engineers. That subtopic (4.2), recommends the use, in conceptual modeling, of notations

that benefit from generalized acceptance, such as the Unified Modeling Language (UML). The use of notations based on Discrete Mathematics is also referred as being advantageous in the analysis of some critical functions or components. In the subtopic 5.1 – *The System Definition Document* – of the topic 5 – *Requirements Specification* – is said that the system definition documentation may include conceptual models to illustrate several aspects of the system, such as workflows. A new mention to formal notations appears in the subtopic 6.3 – *Model Validation* – of the topic 6 – *Requirements Validation* – to add that their use allows the proof of some properties of those specifications.

1.3 Contribution of this dissertation

At this point, it is necessary to make a statement concerning the term *validation*. Although the SWEBOK uses the term validation associated to the application of analysis methods with the aim of proving some properties of the models, that use of the term is not universal among the software engineering scientific community. In this dissertation, we will adopt the interpretation of the colored Petri nets research community; i.e., the term *validation* is used to refer to the actions, taken by the software engineers, to obtain from the users the confirmation that the model really describes their requirements, meaning that the requirements were correctly understood. For the purposes that are described by the SWEBOK, in the subtopic 6.3 of the software requirements KA, we will use the term *verification*.

According to [Aalst, 2004] there are three kinds of analysis that should be accomplished before a workflow is put into production: (1) *validation*, to check if the workflow behaves as expected; (2) *verification*, to study the correctness of a workflow; and (3) *performance analysis*, to estimate the solution conformance with throughput times, service levels, and resource utilization. This dissertation is solely concerned with the first kind of analysis, at the process level; i.e., we are considering, neither the resource dimension (where resources estimation is supposed to be reached) nor the case dimension (where a concrete instance of a workflow process is analyzed, both in its common aspects and in its particularities).

In this dissertation, we describe one proposal that uses CPN-Tools [Lafon *et al.*, 2001] and the BRITNEY Animation tool [BRITNeY] in the generation of interactive animation prototypes to allow stakeholders to be confronted with executable versions of

UML use case and sequence diagrams of previously elicited requirements. This approach towards user requirements validation is illustrated with a real demonstration case where a healthcare information system must be built to support explicitly the interaction between people within a pervasive workflow execution.

1.4 uPAIN demonstration case

The demonstration case considered in this dissertation consists of an information system (uPAIN system) whose main concern is the process of pain control of patients, in a hospital, who are subjected to relatively long periods of pain during post surgery recovery. When a surgery is concluded, the patient enters a recovery period, during which analgesics must be administered to him, in order to minimize the pain that increases as the effects of the anesthesia gradually disappear. This administration of analgesics must be controlled according to a program that depends on factors like some personal characteristics of the patient (weight, age, etc.) and the kind of surgery to which the patient has been submitted. The quantity of administered analgesics must be high enough to eliminate the pain, but low enough to avoid exaggerated or dangerous sedation states. This controlled analgesia is administered to the patient by means of a specialized device called PCA (patient controlled analgesia). PCA is a medication dispensing unit equipped with a pump attached to an intravenous line, which is inserted into a blood vessel in the patient's hand or arm. By means of a simple push button mechanism, the patient is allowed to self administer doses of pain relieving medication (narcotic) on an "as need" basis. This is called a *bolus request*.

The motivation for the development of the uPAIN system arises from the fact that different individuals feel pain and react to it very differently. Moreover, although narcotic doses are predetermined as mentioned previously, there is a considerable variability of their efficacy from patient to patient. This is why anesthesiologists are interested in monitoring several variables, in a continuous manner, during patients' recovery, in order to increase their knowledge on what other factors, besides those already known, are relevant to pain control, and in what measure they influence the whole process. To achieve this, the main idea behind the uPAIN system is to replace the PCA push-button by an interface on a PDA (personal digital assistant), which still allows the patient to request doses from the PCA, but with the addition of the functionality of creating records, in a database, of all those requests, along with other data considered relevant by

the medical doctors, like the values of some predetermined physiological indicators measured by a monitor, and/or other data related to a particular patient's state. Questions concerning symptoms or psychological state may be automatically asked by the system, via the PDA, when the patient requests a dose or at regular time intervals, or even when a medical doctor decides to ask them.

So, the uPAIN system is intended to: (1) provide a platform that enables the registration of patients' pain levels and the occurrence of several symptoms related with the analgesia processes, as frequently as desired; (2) allow the medical staff to be permanently aware of the occurrence of all the relevant facts of the patients' recovery and pain control processes and, (3) allow permanent remote wireless communication among system, patients and medical staff.

1.5 Structure of this document

In this chapter, the subjects of requirements and modeling, as well as their interrelationships were introduced. To illustrate their importance, the perspective of the workflow community on modeling, and the relationship between both subjects reflected by the SWEBOK were pointed-out. The chapter concludes with the presentation of the proposed contribution of this dissertation, along with the introduction to the adopted demonstration case.

The remainder of this dissertation is organized as follows. Chapter 2 introduces and defines the structure, behavior, properties and analysis methods of low-level Petri nets. It also points-out the limitations of low-level Petri nets, as a starting point to briefly present their most important extensions and high-level Petri nets. Chapter 3 presents a proposal of a technique to derive Colored Petri Nets (CP-nets) from UML models of functional user requirements, with the intent of applying them to control interactive animation prototypes, for validation of those requirements by stakeholders. Chapter 4 describes how that technique, CPN-Tools and the BRITNeY Animation tool are recommended to be applied to support the building of an animation prototype for the uPAIN demonstration case.

Chapter 2

Petri Nets Concepts

To address concurrency in systems, Carl Adam Petri introduced, in his PhD dissertation *Komunikation mit Automata* [Petri, 1962], presented in 1962, at Darmstadt, Germany, a special class of generalized graphs, now called Petri nets. Because they have a mathematical description associated to the graphical representation, Petri nets are a tool that is well suited not only for modeling but also for analysis and study of discrete event systems (DES), especially those in which events can occur concurrently. In other words, using Petri nets to model a system leads to a mathematical description of that system, which allows the analytical study of its structure and properties, including those associated to simultaneous occurrence of the system's events.

2.1 Petri nets structure

Every graph consists of two types of elements – *vertices* or *nodes*, and *edges* – and a given interconnection of these elements (Fig. 2.1a). Formally,

> a **graph** is a triple $G = (V,E,\phi)$, where V is a nonempty set of nodes, E is a set of edges, and ϕ is a mapping from the set of elements of E to a set of pairs of elements of V.

An edge is said to be *directed* if the connection it establishes between a pair of nodes is ordered. To indicate the direction, an arrow is placed on the edge (Fig. 2.1b). A *directed graph* is a graph where all the edges are directed. **Petri nets are directed graphs**.

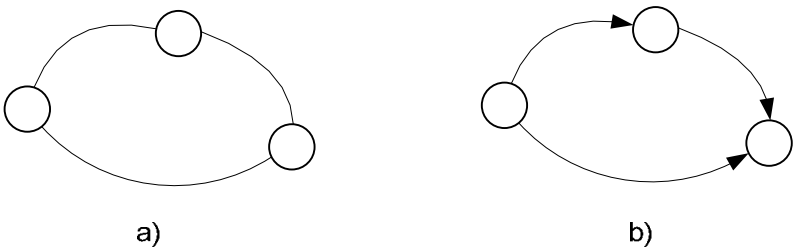

Fig. 2.1 – Examples of graphs: a) undirected, and b) directed.

In a graph, two nodes that are connected by an edge are called *adjacent* nodes. A graph is called a *multigraph* if it contains parallel edges, i.e., edges that connect the same pair of nodes and, if directed, have the same direction (Fig. 2.2a). **Petri nets are multigraphs**. When a Petri net has no more than one *arc* (the name given to edges in Petri nets) connecting two nodes, then it is called an *ordinary* Petri net.

In order to simplify the graphical representation of parallel arcs, it is better to replace them by a single arc with a non-negative integer attached to it (called *weight*) representing the number of parallel arcs. The concept of weight corresponds to a generalization of the notion of arc that is most adequate for mathematical representations. Naturally, arcs with weight 0 are not represented graphically, because a weight with the value 0 means the inexistence of an arc. Furthermore, as a convention, when the weight is 1 the corresponding arc is drawn without the number 1 attached to it (Fig. 2.2b). Therefore, in the graphical representation of ordinary Petri nets there are no integers attached to arcs.

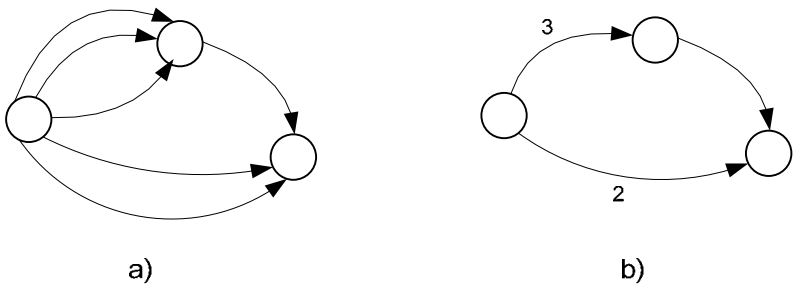

Fig. 2.2 – Representation of a multigraph: a) without the use of weights, and b) using weights.

Another characteristic of Petri nets is that **they are *bipartite* graphs**. This means that they have two types of nodes and that an arc cannot directly connect nodes of the same type. In a Petri net, the two types of nodes are called *places* and *transitions*.

Therefore, because Petri nets are bipartite graphs, an arc always connects from a place to a transition or from a transition to a place and never from a place to another place or from a transition to another transition.

Fig. 2.3 shows an example of an ordinary Petri net. Conventionally, places are represented as circles, transitions as black filled rectangles (or thick bars), and arcs are the directed lines connecting the places and the transitions. However, depending on the type of Petri net, and personal preferences, these elements can have different configurations. Therefore, transitions can be represented as unfilled rectangles or just thin bars, places can appear as ellipses, and arcs can always be shaped freely, according to personal taste and readability concerns.

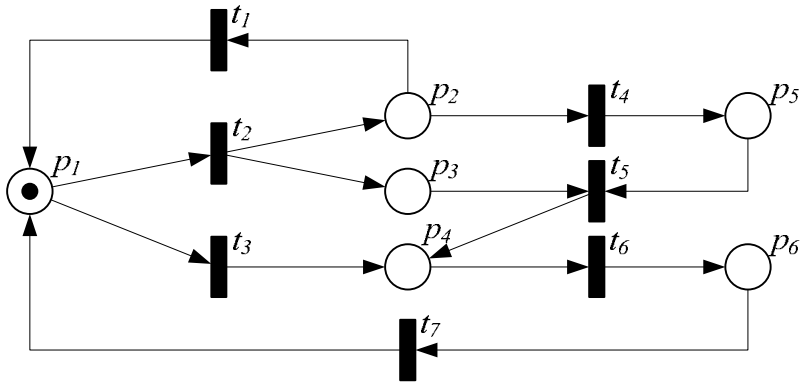

Fig. 2.3 – An example of an ordinary Petri net.

Given the concepts just presented, it is now possible to consider the following formal definition:

> A **Petri net** is a bipartite directed graph represented by a quadruple $N = (P,T,I,O)$, where,
>
> $P = \{p_1,p_2,...,p_n\}$ is a finite set of places,
>
> $T = \{t_1,t_2,...,t_m\}$ is a finite set of transitions,
>
> $I(p, t)$ is a mapping $P \times T \rightarrow \mathbb{N} \cup \{0\}$, corresponding to the set of the weights of the arcs directed from places to transitions, and

$O(p, t)$ is a mapping $P \times T \rightarrow \mathbb{N} \cup \{0\}$, corresponding to the set of the weights of the arcs directed from transitions to places.

Instead of using the generalization provided by the notion of weight of an arc, some authors (e.g. [Peterson, 1981]) have preferred to maintain the notion of multiple parallel arcs, and express the sets I and O as being I: $T \rightarrow P^{\infty}$ and O: $T \rightarrow P^{\infty}$ mappings from transitions into bags of places. Naturally, without the use of the weight concept, the definitions used by those authors assume a different form, but they are, of course, perfectly equivalent to the ones used here.

For a given transition t_j, its *preset*, denoted by $^{\circ}t_j$, is the set of its' *input places*, which are the places connected to it by its' *input arcs*, according to the definition

$$^{\circ}t_j = \{p_i \in P: I(p_i,t_j) \neq 0\}$$

Analogously, the *postset* of t_j, denoted by t_j°, is the set of its' *output places*, which are the places connected to it by its' *output arcs*, according to the definition

$$t_j^{\circ} = \{p_i \in P: O(p_i,t_j) \neq 0\}$$

The same notation applies to places, originating the corresponding definitions of the preset (the set of the input transitions) and the postset (the set of output transitions) of a given place p_i

$$^{\circ}p_i = \{t_j \in T: O(p_i,t_j) \neq 0\}, \qquad p_i^{\circ} = \{t_j \in T: I(p_i,t_j) \neq 0\}.$$

For example, for the Petri net of Fig. 2.3, the presets and postsets of place p_1 and transition t_2 are

$$^{\circ}p_1 = \{t_1,t_7\}, \quad p_1^{\circ} = \{t_2,t_3\}, \quad ^{\circ}t_2 = \{p_1\}, \quad t_2^{\circ} = \{p_2,p_3\}$$

Naturally, the enumeration of the set of places, the set of transitions, and the presets and postsets of all the places (or all the transitions) of an ordinary Petri net, constitute a complete mathematical definition of its structure.

2.1.1 Some structural particularities

A node is called a *source* if it has only outgoing arcs (its' preset is $\varnothing$). A node which has only incoming arcs (its' postset is $\varnothing$) is called a *sink*.

A *path* is a set of k arcs and $k+1$ nodes, with $k \in \mathbb{N}$, such that, for $i \in \mathbb{N}$ and $1 \leq i \leq k$, the i^{th} arc either connects the i^{th} node to the $i+1^{th}$ node, or the $i+1^{th}$ node to the i^{th} node. If, for all $1 \leq i \leq k$, the i^{th} arc connects the i^{th} node to the $i+1^{th}$ node, the path is called a *directed path*. A path in which every arc is traversed only once is called a *simple path*. A path in which every node is traversed only once is called an *elementary path*. It is important to note that an elementary path is always a simple path, but a simple path may not be an elementary path.

A Petri net is *connected* if and only if there is a path (not necessarily directed) from any node to any other node. A Petri net is *strongly connected* if and only if there is a directed path from any node to any other node. Naturally, a Petri net that has a source or sink cannot be strongly connected.

A *directed circuit* is a directed path from one node back to itself. A *directed elementary circuit* is a directed circuit in which no node appears more than once. Because they are adequate for modeling sequences of events, directed elementary circuits play a key role in the performance analysis of DES modeled with Petri nets [Cassandras, 1993].

For certain purposes, it is often useful to consider only one part of a Petri net, called a *subnet*, defined as follows:

A *subnet* of a Petri net $N = (P,T,I,O)$ is a Petri net $N_s = (P_s,T_s,I_s,O_s)$, where

$$P_s \subseteq P, \quad T_s \subseteq T, \quad I_s = (P_s \times T_s) \cap I, \quad O_s = (P_s \times T_s) \cap O$$

2.2 Petri nets behavior

Petri nets are also capable of modeling the dynamics of a system. Yet, the elements of Petri nets defined so far are used to model only the structure of a system. In order to model the system's dynamics an extra element of the Petri nets meta-model is needed – the *token*. Tokens are used to express the states of a system in a Petri net, and are represented in the graph as black dots that can reside in places. In the example of Figure 2.1,

the particular state shown for the net is described with only one token in place p_1. When the allowed number of tokens in a place is limited, the net is said to have *capacities*. Each possible distribution of tokens in a net that respects those limits is called a *marking*, that is,

> For a given Petri net $N = (P,T,I,O)$, a mapping M: $P \rightarrow \mathbb{N} \cup \{0\}$ is a marking of N iff
>
> $$\forall\, p \in P, M(p) \leq K(p)$$
>
> where $M(p)$ is the number of marks in place p and $K(p)$ is the capacity of place p.

This way, each possible state of the system is represented, in its Petri net model, by a particular marking. Therefore, in order to represent the transition from one given state of the system to a subsequent state, there must be a process by which the Petri net model of the system changes from the corresponding first marking into the other. This process is called *firing*, and is the mission of the nodes that, for that reason, were very appropriately named “transitions”.

Firing consists of the action through which transitions remove tokens from their input places and add tokens to their output places. When a transition fires, the number of tokens removed from an input place equals the weight of the corresponding input arc. Similarly, the number of tokens added to an output place equals the weight of the corresponding output arc. However, for a transition to be allowed to fire, it must be *enabled*, according to the definition:

> A transition t is *M-enabled* (enabled by a given marking M), iff
>
> $$\forall p \in {}^\circ t, M(p) \geq I(p, t) \wedge \forall p \in t^\circ, M(p) \leq K(p) - O(p,t)$$

Therefore,

> When a transition t is *M_0-enabled*, it can *fire*, originating $M_1(p)$, which is a *next marking* of $M_0(p)$, given by
>
> $$\forall p \in P, M_1(p) = M_0(p) - I(p,t) + O(p,t)$$

It is then said that *t fired from M_0 to M_1*, expressed with the notation $M_0[t\rangle M_1$.

2.3 Petri nets and Linear Algebra

Because many aspects of Petri nets can be represented as vectors and matrices, Linear Algebra, due to its strong adequacy for implementation in computer programming, plays a very important role in Petri net theory.

The topology, or structure, of a Petri net can be represented by an integer matrix. However, for that purpose, the Petri net may not have *self-loops*, i.e., it must be *pure*.

A Petri net $N = (P,T,I,O)$ is *pure*, iff

$$\forall p \in P,\ t \in T,\ I(p,t) \cdot O(p,t) = 0$$

If, for a given pair of nodes (one place p_i and one transition t_j), $I(p,t) \cdot O(p,t) > 0$, this means that there is an arc directed from p_i to t_j and an arc directed from t_j to p_i, forming what is called a self-loop, and then, the Petri net is not pure.

The matrix representing the structure of a pure Petri net with n places and m transitions is an $n \times m$ matrix, called *incidence matrix* (or flow matrix).

The elements of the *incidence matrix* $\underline{C}$, of a pure Petri net $N = (P,T,I,O)$, with $n = \#P,\ m = \#T,$ are given by:

$$C_{ij} = O(p_i,t_j) - I(p_i,t_j),\ \text{for}\ 1 \le i \le n\ \wedge 1 \le j \le m$$

Observing this definition, it is evident that it completely describes the structure of a pure Petri net $N = (P,T,I,O)$, as follows:

$\#P$ equals the number of rows of $\underline{C}$,

$\#T$ equals the number of columns of $\underline{C}$,

$I(p_i,t_j) = -\min\{C_{ij},0\}$,

$O(p_i,t_j) = \max\{C_{ij},0\}$.

From this, it becomes evident that a self-loop would not be represented in the incidence matrix $\underline{C}$, because the corresponding element in the matrix would not allow the

reconstruction of the original arcs. For a self-loop formed by arcs of equal weight, the resulting element of $\underline{C}$ would be zero, eliminating the traces of the self-loop. For a self-loop with arcs of different weights, the resulting element in $\underline{C}$ would lead to the erroneous interpretation that only an arc, with a weight equal to the difference of the original weights, existed between the two nodes.

As an example, the incidence matrix $\underline{C}$, and the corresponding I and O mappings in the matrix form, for the Petri net of Fig. 2.3 are:

$$\underline{C} = \begin{bmatrix} 1 & -1 & -1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

$$\underline{I} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \underline{O} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Using a vector $M_i = [M(p_1)\ M(p_2)\ M(p_3)\ \ldots\ M(p_n)]^T$ to represent a marking, and another vector S in which all elements are zero, except the j^{th} element, which is 1, corresponding to a transition t_j enabled by M_i, the next marking M_{i+1} is obtained by the equation

$$M_{i+1} = M_i + \underline{C} \cdot S \qquad (1)$$

where $\underline{C}$ is the incidence matrix.

2.4 Analyzing Petri nets

From what has just been stated, the enabling of a transition only depends on the marking of the places that are directly connected to it. Furthermore, a transition only needs to be enabled in order to be allowed to fire. So, it becomes immediately apparent that in a given state (a given marking) of a Petri net, two transitions that have disjoint presets and disjoint postsets may be concurrently enabled and, consequently, they may fire simultaneously. This natural support of concurrency is clearly a major advantage of using Petri nets as a modeling tool.

Besides, being mathematically defined, Petri net models describe systems in an unambiguous manner, thus preventing confusion when used for communication purposes among people involved in the study of the systems they describe. But Petri nets have additional strengths that result from the fact that they are a mathematical meta-model. Through mathematical descriptions, it is possible to express particular structures and properties of Petri nets, as well as using analysis algorithms to detect and verify those structures and properties.

2.4.1 Conflict and confusion

One of the powers of Petri nets is unquestionably the support for concurrency. Yet, the same characteristic that makes concurrency possible – the capability for two transitions to be able to fire simultaneously – may bring undesired situations, like conflicts or confusion.

> A *conflict* occurs when concurrently enabled transitions share a common input place, i.e., iff for a Petri net $N = (P,T,I,O)$,
>
> $$\forall p \in P,\ \#p^\circ > 1$$

For example, in the Petri net of Fig. 2.3 transitions t_1 and t_4 both share the same input place p_2, i.e.

$$p_2^\circ = \{t_1, t_4\}$$

The same happens with t_2, t_3 and p_1

$$p_1^\circ = \{t_2, t_3\}$$

These conflicts are easily identified by observing the matrices $\underline{I}$ or $\underline{C}$. The rows 1 and 2, corresponding to places p_1 and p_2, both have more than one positive value, in I (the elements $I(1,2)$, $I(1,3)$ and $I(2,1)$, $I(2,4)$). Naturally, in $\underline{C}$ that corresponds to the existence of more than one negative element in the same rows (the elements $C(1,2)$, $C(1,3)$ and $C(2,1)$, $C(2,4)$).

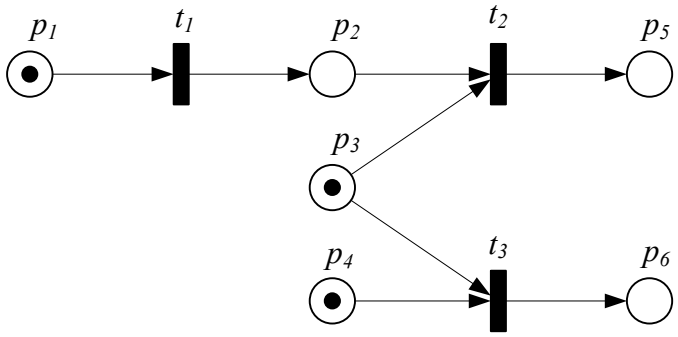

Fig. 2.4 – An example of confusion.

In the Petri net of Fig. 2.4, only t_1 and t_3 are enabled. If t_3 fires first, then t_1 can fire and the execution stops. However, if t_1 fires first, then t_2 and t_3 become both enabled but in conflict due to the common input place p_3. In cases like this one, in which the occurrence of a conflict depends on firing sequences, we say we are in the presence of *confusion*.

2.4.2 Reachability and coverability

Reachability concerns the question whether a given marking of a Petri net is possible. From the modeled system point of view, we might be interested in checking if it is possible that a given state can be reached, whether because it is forbidden, or mandatory, and under which circumstances. It might be possible for the system to reach a given state, only when starting from a particular initial state, and after a particular sequence of events. If we know in advance that an already existing system, under certain conditions, reaches a certain state, we might be interested in checking if a Petri net we have built to model it reflects that reality. Alternatively, we might be interested in assuring that a system that is to be built, according to a Petri net model, will reach (or not) a given state, under certain conditions.

If, from an initial marking M_0, of a Petri net N, a transition t_1 fires, producing a next marking M_1, i.e., $M_0[t_1 \rangle M_1$, then we say that M_1 is *immediately reachable* from M_0. If

another marking M_2 is immediately reachable from M_1, through the firing of another transition t_2, that is, $M_1[t_2\rangle M_2$, then we say that M_2 is *reachable* from M_0. We can then write $M_0[t_1t_2\rangle M_2$, or $M_0[F_S\rangle M_2$, where $F_S = t_1t_2$ is called a *firing sequence*, and represents the sequence of transitions that were fired to reach M_2 from M_0. The set of all the possible firing sequences from M_0 may be denoted by $F_S(N,M_0)$.

> The *reachability set of N from* M_0, denoted by $R(N,M_0)$, is the smallest set of reachable markings of a Petri net $N = (P,T,I,O)$, with initial marking M_0, such that:
>
> $$\begin{cases} M_0 \in R(N,M_0) \\ M_i \in R(N,M_0) \ \wedge \ \exists\, t_k \in T:\ M_i[t_k\rangle M_j \ \Rightarrow \ M_j \in R(N,M_0) \end{cases}$$

A Petri net $N = (P,T,I,O)$ for which an initial marking M_0 is reachable from any marking $M_i \in R(N,M_0)$ is said to be *reversible*. A less demanding criterion for reversibility, adequate to a larger number of situations, is for a marking $M_j \in R(N,M_0)$, called an *intermediate state*, to be reachable from any other marking $M_i \in R(N,M_0)$. When any marking $M_i \in R(N,M_0)$ is reachable from any other marking $M_j \in R(N,M_0)$, we say that N is *completely controllable*.

At this point, it is interesting to note that, for pure Petri nets, if a given firing sequence is possible from an initial marking M_0, equation (1), of page 18, may be used to obtain the resulting marking. For this purpose, the elements of the vector S are zero for the transitions that are not in the firing sequence, and are equal to the number of times a transition fired, for the elements corresponding to transitions belonging to the firing sequence. Yet, this method should be used with caution, because some information (namely the order of the firings) is lost in the process of converting the firing sequence into the vector S, and that fact might lead to incoherent results.

A variation of the question of reachability is the case in which we are interested in knowing if it is possible to reach, from a given initial marking M_0, a marking M_i for which the individual markings of the places satisfy predetermined minimum values. This is *the coverability problem*. We might be interested in such an analysis in the case of models for which some places represent counters. Naturally, in this type of analysis it is possible to have more than one marking that satisfies that condition.

In formal terms, in a coverability problem we are interested in determining if, given a Petri net $N = (P,T,I,O)$, a marking $M_i \in R(N,M_0)$, and another marking M_j, M_i covers M_j, i.e., $M_i \geq M_j$, where $M_i \geq M_j$ is defined by:

$$M_i \geq M_j \Leftrightarrow \forall p \in P,\ M_i(p) \geq M_j(p)$$

2.4.3 Boundness and safeness

Some systems restrict the number of tokens in a given place to a certain limit. That is the case, for instance, of a place that is modeling a storage space for limited units of a resource, or a place modeling a counter of a system to be implemented in hardware or software. Those places must be bounded for the initial marking, according to the following definition:

> Given a Petri net $N = (P,T,I,O)$, we say that a place p is *k-bounded*, for an initial marking M_0, iff
>
> $$\forall\ M_i \in R(N,M_0),\ \exists\ k \in \mathbb{N}: M_i(p) \leq k$$

We say that a Petri net is *k-bounded* for a given initial marking, if all its places are *k*-bounded for that initial marking, i.e.

> A Petri net $N = (P,T,I,O)$ is *k-bounded*, for an initial marking M_0, iff
>
> $$\forall\ p \in P,\ M_i \in R(N,M_0),\ \exists\ k \in \mathbb{N}: M_i(p) \leq k$$

These definitions present boundness as being dependent of the initial marking of the Petri net, but it is possible for a Petri net to have a topological structure that guaranties boundness for any initial marking. In this case, we say that the Petri net is *structurally bounded*.

When a Petri net is *1-bounded*, i.e., when $k = 1$, we say that it is *safe*.

2.4.4 Conservation

Petri nets are often used to model resource allocation systems. For instance, a Petri net modeling the allocation of a pool of printers, shared among several users, would have a certain number of tokens corresponding to the number of printers in the pool. These tokens would be located in several places, according to their condition of readi-

ness for use. One place could be used to indicate that a printer is ready, another place would indicate that the printer is busy, other places could represent other states, like out of paper, out of toner, on maintenance, etc. Because the number of printers in the pool is a constant, the number of tokens in the Petri net should remain constant, independently of their distribution among the places. To this requirement, we call *conservation*.

> A Petri net $N = (P,T,I,O)$, with an initial marking M_0 is *strictly conservative* iff
>
> $$\forall\, p \in P,\ M_i \in R(N,M_0),\ \Sigma\, M_i(p) = \Sigma\, M_0(p)$$

According to this definition, in order for a Petri net to be strictly conservative, the firing of one transition cannot change the total number of tokens in the net. Therefore, each transition has to produce as many tokens as it consumes. However, Petri nets modeling finite resources allocation systems often have places whose tokens do not represent resources. These places may represent counters, conditions that determine the changes in the resource states, etc. In such cases, and for the majority of the applications, strict conservation may not be required, and it can be allowed for some transitions to produce a different number of tokens than the ones they consume. A solution to obtain a definition of conservation that supports these cases is to consider, not the simple sum of the numbers of tokens in the places, but a weighted sum. By assigning a non-negative rational weight (not necessarily integer) to each place, it may be possible to obtain a constant weighted sum after each transition firing, allowing the introduction of a different definition for conservation:

> A Petri net N = (P,T,I,O), with an initial marking M_0 is conservative with respect to a weighting vector $\mathbf{w} = (w_1, w_2, \ldots, w_n)$, where n = #P, iff
>
> $$\forall\, p_i \in P,\ M_j \in R(N,M_0),\ \Sigma\, [w_i \cdot M_j(p_i)] = \Sigma\, [w_i \cdot M_0(p_i)]$$

It is important to note that conservation with respect to a weighting vector may result from the occurrence of strict conservation of a subnet. That is the case when the elements of the weighting vector have only values that are either 0 or 1. This means that the elements that are equal to 0 correspond to places that do not support conservation, while the elements that are equal to 1 correspond to the places belonging to a strictly conservative subnet. If this subnet corresponds to the part of the Petri net that models

the finite resource allocation, then this conservation criterion may satisfy the modeler's needs in what concerns the verification of the model.

2.4.5 Net invariants

Net *invariants* are one of the structural properties of Petri nets, i.e., properties that depend only on the topological structure of the Petri net and not on the net's initial marking. Therefore, because they are structural properties, invariants are important means for analyzing Petri nets since they allow the investigation of the net's structure independently of any dynamic process. Moreover, that analysis can be performed on local subnets without considering the whole system. Invariants are also used for model verification.

There are two kinds of invariants: *place invariants* and *transition* invariants.

Place invariants

The notion of place invariant coincides with the particular case in which all the elements of the vector of weights used to define conservation are integer, i.e., a *place invariant* (or *P-invariant*) is a non-negative integer vector, with dimension equal to the number of places in the Petri net. The i^{th} element is a weight associated to the i^{th} place, such that the weighted sum of markings of the places is a constant for any firing sequence. However, the preferred definition for a *P*-invariant has a form that leads to an algorithm for their determination:

> A *P-invariant* of a Petri net $N = (P,T,I,O)$, with $\#P = n$ and an incidence matrix $\underline{C}$, is a non-negative integer vector $\boldsymbol{x}$, with n elements, that satisfies the relation
>
> $$\underline{C}^{T} \cdot \boldsymbol{x} = 0 \qquad (2)$$

The set of places corresponding to the strictly positive elements of $\boldsymbol{x}$ is called the *support* of the invariant [Levis, 1989], and is denoted $<\boldsymbol{x}>$. When the support of an invariant does not contain the support of another invariant, except itself and the empty set, the support is said to be *minimal*.

By multiplying the equation (1) by $\boldsymbol{x}^{T}$ and by using (2) to eliminate the term $\boldsymbol{x}^{T} \cdot \underline{C} \cdot S$, we obtain (3), which establishes the relationship between (2) and the conservation property, as stated by the theorem:

The vector $\boldsymbol{x}$, is a *P*-invariant of a Petri net $N = (P,T,I,O)$, iff

$$\forall\ M_0,M_i \in R(N,M_0),\quad \boldsymbol{x}^{\mathrm{T}} \cdot M_i = \boldsymbol{x}^{\mathrm{T}} \cdot M_0 \qquad (3)$$

This theorem establishes the conservation of the number of tokens belonging to the support $<\boldsymbol{x}>$, weighted by the value of the elements of the *P*-invariant vector $\boldsymbol{x}$.

A *P-component* is the subnet associated with a *P*-invariant, according to the definition:

> The *P-component* $[\boldsymbol{x}]$ of a Petri net $N = (P,T,I,O)$, having a *P*-invariant $\boldsymbol{x}$ with the support $<\boldsymbol{x}>$, is a subnet of N whose set of places is $<\boldsymbol{x}>$ and whose transitions are the input and output transitions of the places of $<\boldsymbol{x}>$, i.e.,
>
> $$[\boldsymbol{x}] = (P_x,T_x,I_x,O_x) :$$
>
> $$P_x = <\boldsymbol{x}>,$$
>
> $$T_x = \{{}^\circ p : p \in P_x\} \cup \{p^\circ : p \in P_x\},$$
>
> $$I_x = (P_x \times T_x) \cap I,$$
>
> $$O_x = (P_x \times T_x) \cap O.$$

Having in consideration the discussion of the last paragraph of the topic on conservation, it becomes evident that the determination of invariants is a sustained method for the verification of a subnet that is responsible for the modeling of finite resources. If a *P*-component is found which coincides with that subnet, then that subnet satisfies the conservation property required for the modeling of the finite resources allocation.

Transition invariants

The notion of transition invariant, or *T-invariant*, of a Petri net is associated with a set of transitions that, together with their presets and postsets, form a cyclic part of the Petri net. It indicates which transitions must fire and how many times each, so that an initial marking is repeated.

A *T-invariant* of a Petri net $N = (P,T,I,O)$, with $\#T = m$ and an incidence matrix $\underline{C}$, is a non-negative integer vector $\boldsymbol{y}$, with m elements, that satisfies the relation

$$\underline{C} \cdot \boldsymbol{y} = 0 \qquad (4)$$

The interpretation of the vector $\boldsymbol{y}$ is that it contains positive integers in the positions corresponding to the transitions belonging to the transition invariant, and zeros everywhere else. Each positive integer denotes how many times the corresponding transition must fire in order that the initial marking is repeated.

In an analogous way as has been done with *P*-invartants, the support $<\boldsymbol{y}>$, of a *T*-invariant $\boldsymbol{y}$, is the set of transitions corresponding to the strictly positive elements of $\boldsymbol{y}$ [Levis, 1989], and the corresponding *T*-component $[\boldsymbol{y}]$ is a subnet whose set of transitions is $<\boldsymbol{y}>$ and whose places are the input and output places of the transitions of $<\boldsymbol{y}>$.

2.4.6 Deadlock and liveness

A *deadlock* is a marking in which none of the transitions of a Petri net is enabled and, therefore, none of them can fire, causing the execution of the Petri net to end. Because deadlocks may correspond to anomalies in the system design or in its' Petri net model, or may be natural behaviors of a system which should be avoided or require special attention, the causes of deadlocks have been extensively studied. The property of Petri nets related to deadlocks is liveness.

A transition t_i, of a Petri net N, is *live*, for an initial marking M_0, if, for every reachable marking $M_j \in R(N,M_0)$, a firing sequence from M_j exists which contains t_i. A Petri net is live for an initial marking M_0 if all its transitions are live for M_0.

Real systems tend to be complex, originating large and complex Petri net models that make the analysis of liveness very expensive – in terms of computational resources – and sometimes unfeasible. For that reason, by relaxing some conditions, different levels of liveness have been defined. So, for a Petri net N, with initial marking M_0, a transition t_i is said to be:

L0-live – or dead – if t_i can never be fired.

L1-live – potentially fireble – if t_i can be fired at least once, i.e., if a marking $M_j \in R(N,M_0)$ exists, such that t_i is enabled in M_j.

L2-live – if, for a positive integer k, t_i can be fired at least k times in one of the firing sequences of $S(N,M_0)$.

L3-live – if t_i appears an infinite number of times in some of the firing sequences of $S(N,M_0)$.

L4-live – or live – if t_i is L1-live for all markings of $R(N,M_0)$.

A Petri net is said to be *Lk-live*, with $k \in \{0,1,2,3,4\}$, if all its transitions are Lk-live. An L1-live Petri net may also be called *almost-live*.

It is important to note that liveness is closely related to reachability. If the liveness of a transition depends on its capability for firing, it depends on the reachability of a marking that enables it. So, the complexity of both types of analysis – liveness and reachability – is equivalent.

2.4.7 Reachability and coverability trees and graphs

As invariants, the *reachability and coverability trees and graphs* are analysis tools that reflect the structure of a Petri net and allows the study of its behavior. As the name suggests, a reachability tree is a graph that represents the reachability set of a Petri net.

Each node of a reachability tree is a marking, being the first node the initial marking. Each transition that is enabled by a given marking is represented by a directed arc, connecting the node corresponding to that marking to the node corresponding to the marking that results from its firing.

Fig. 2.6 shows a Petri net with initial marking $M_0 = (1,0,0,0,0,0)$ and, in a), the reachability tree for that initial marking. In b) is shown the equivalent *reachability graph*. As can be seen, the reachability graph differs from the reachability tree in the fact that there are no repeated nodes, originating a more compact representation of the reachability set.

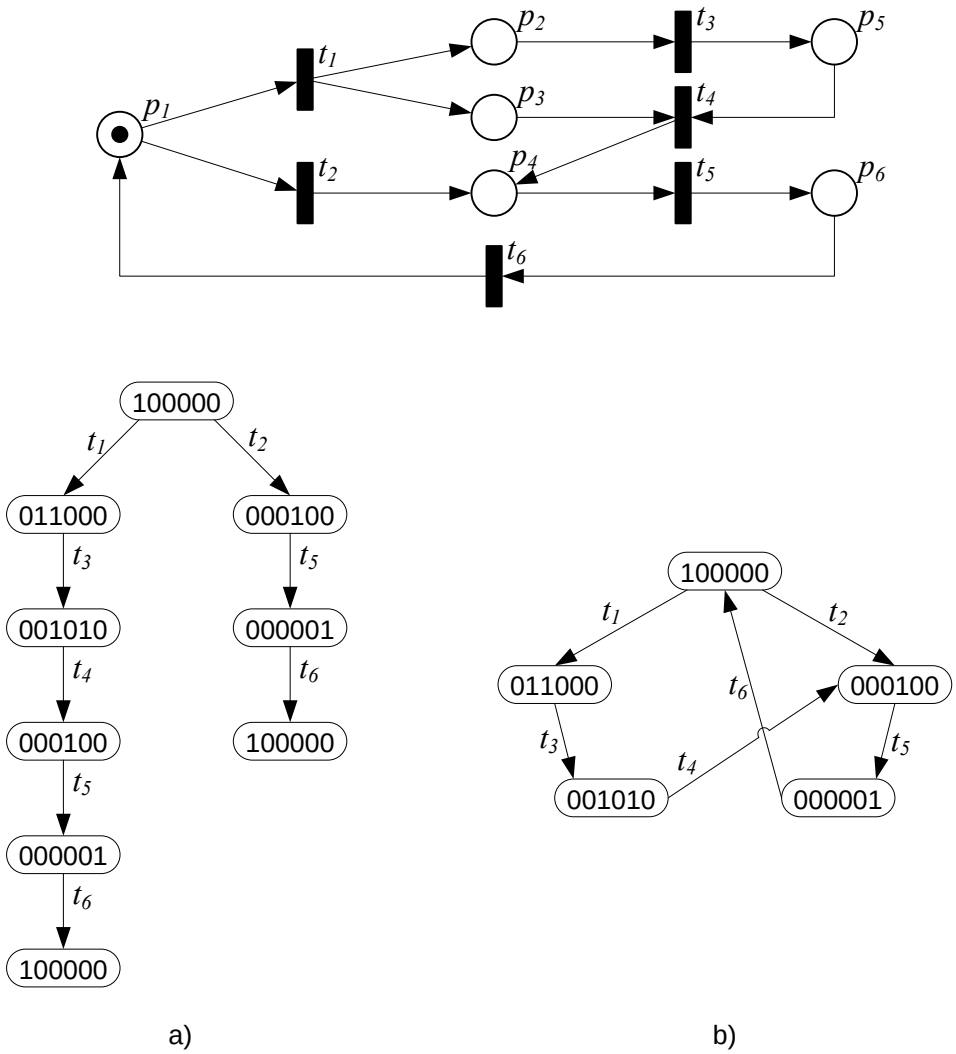

Fig. 2.6 – A marked Petri net with a) its reachability tree and b) its reachability graph.

In section 2.3.1, page 6, a marking M was defined as a mapping of the type

$$M: P \rightarrow \mathbb{N} \cup \{0\}$$

This means that the values of $M(p)$ are non-negative integers. However, it is possible that the number of tokens in a place grows indefinitely. When that is the case, the symbol ω is used to represent an arbitrarily large number of tokens, justifying the use of the form

$$M: P \rightarrow \mathbb{N} \cup \{0,\omega\}$$

for the definition of a marking. The symbol ω – which may be interpreted as infinity – satisfies the following properties:

$$\forall\, a \in \mathbb{N},\ \omega + a = \omega \ \wedge\ \omega - a = \omega \ \wedge\ a < \omega \ \wedge\ \omega \leq \omega$$

In the Petri net of Fig. 2.3, place p_3 is an example for the use of the symbol ω. For the initial marking shown, the firing of t_2 corresponds to

$$(1,0,0,0,0,0)\ [t_2\rangle\ (0,1,1,0,0,0)$$

If t_1 fires next,

$$(0,1,1,0,0,0)\ [t_1\rangle\ (1,0,2,0,0,0)$$

And if t_2 fires again,

$$(1,0,2,0,0,0)\ [t_2\rangle\ (0,1,3,0,0,0)$$

As this initial sequence shows, if these two transitions – t_1 and t_2 – keep firing alternately, the number of tokens in p_3 will grow indefinitely, the same happening to the number of different markings for the Petri net. For these cases, it is impossible to build a finite reachability tree.

The symbol ω allows to represent, with a single tuple, a Petri net marking that covers any possible marking where the number of tokens in a given place may be arbitrarily large. While some authors, like [Peterson, 1981], maintain the designation *reachability tree* even for graphs where the symbol ω appears, i.e., that use the concept of coverability, other authors, like [Murata, 1989], use that designation only for bounded Petri nets, and prefer the designations *coverability tree*, or *coverability graph* [Reisig, 1985a], for graphs representing the reachability sets of unbounded Petri nets.

The algorithm for the construction of the coverability tree, in [Peterson, 1981], classifies each node as frontier, terminal, duplicate, or internal, and, with slight adaptations, consists of the following:

1. Define the initial marking as root of the tree and, initially, as a frontier node.
2. While frontier nodes remain, do:

    2.1. Select a frontier node *x*.

2.2. If there exists another node y in the tree which is not a frontier node, and has the same marking associated with it, i.e., $M_x = M_y$, then redefine node x as a *duplicate* node.

2.3. If no transitions are enabled for the marking M_x, then redefine node x as a *terminal* node.

2.4. If there are enabled transitions for the marking Mx, then:

2.4.1. Redefine node x as an *interior* node.

2.4.2. For all transitions t_j which are enabled for the marking M_x, do:

2.4.2.1. Create a new node z in the coverability tree.

2.4.2.2. Draw an arc, labelled t_j, directed form node x to node z.

2.4.2.3. Define node z as a frontier node.

2.4.2.4. For each place p_i, the marking $M_z(p_i)$ is:

a) If $M_x(p_i) = \omega$, then $M_z(p_i) = \omega$.

b) If there exists a node y on the path from the root to node x such that $M_y \neq M_z \wedge M_z \geq M_y \wedge M_z(p_i) > M_y(p_i)$, then $M_z(p_i) = \omega$.

c) Otherwise, $M_z(p_i) = M_x(p_i) - I(p_i,t_j) + O(p_i,t_j)$.

This algorithm ends, because the reachability tree is guaranteed to be finite [Peterson, 1981]. Fig. 2.7 shows an unbounded Petri net and its coverability tree obtained according to this algorithm.

The coverability tree allows several types of analysis. Naturally, being based on the coverability property, determining if a given marking is covered is done by mere observation of the coverability tree. Likewise, the determination of safeness and boundness are quite intuitive. If only 0 and 1 appear in the coverability tree, the Petri net is obviously safe. The Petri net is bounded if and only if the symbol ω never appears in the coverability tree. If this happens, the Petri net represents a finite state system, and the coverability tree constitutes a state graph that allows answering any question related to boundness [Peterson, 1981]. For unbounded Petri nets, it is still possible to determine, by exhaustive observation of the coverability tree, the bounds for the places that do not have the symbol ω in the reachability tree. Conservation with respect to a weighting

vector may also be tested with the reachability tree. If a node in the reachability tree presents a symbol ω for a place whose corresponding weight is positive, the Petri net is not conservative with respect to that vector. For a bounded Petri net with n places, with k nodes in the reachability tree, it is possible to construct a system of k linear equations in $n + 1$ unknowns (corresponding to the n weights of the vector and the constant weighted sum) for whose solution several algorithms exist [Peterson, 1981].

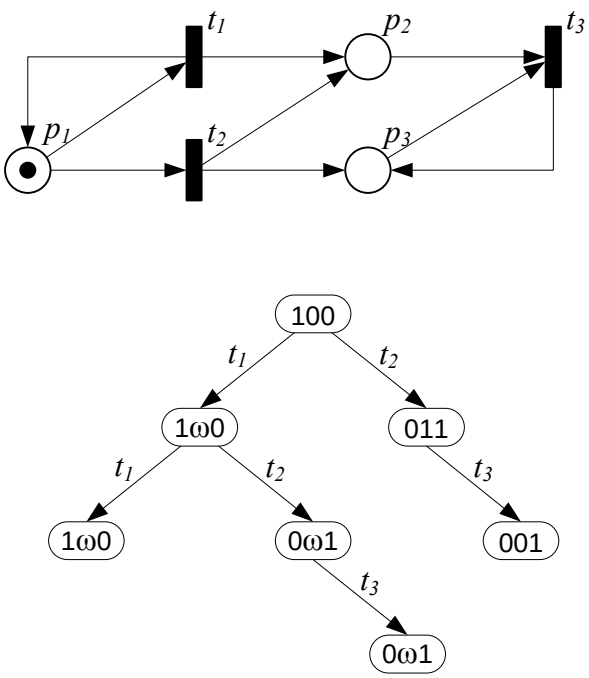

Fig. 2.7 – An unbounded Petri net and its coverability tree.

Although the coverability graph allows several kinds of analysis, including some reachability and liveness questions, it is not powerful enough to answer all reachability or liveness questions or to determine which firing sequences are possible. The presence of the symbol ω corresponds to a loss of information that limits those kinds of analysis. It hides, for example, the fact that a place can only have an odd or an even number of tokens, or an increasing or decreasing number of tokens [Murata, 1989]. For example, in the Petri net of Fig. 2.7, if t_2 fires after some firings of t_1, the number of tokens in p_2 increases 1 unit, and then t_3 fires but the number of tokens in p_2 decreases another unit, yet, the symbol ω for the place p_2 in the nodes of the reachability tree hides that fact.

2.4.8 Reduction and decomposition techniques

The analysis techniques just discussed, based on matrix equations approaches and the coverability tree, have proven to be effective in determining properties of safeness, boundness, conservation and coverability, as well as in establishing a necessary condition for reachability. However, for real systems models, these techniques become too demanding in terms of processing resources, especially for analysis problems concerning reachability or liveness.

In order to reduce the processing cost of analysis of complex Petri nets, reduction and decomposition techniques have been developed. These techniques transform a Petri net in a simpler one (i.e., with fewer nodes). Because this new net is simpler and smaller, it requires a smaller processing effort in the application of analysis techniques.

There is a considerable diversity of reduction techniques, the simplest of which consisting of fusion of places or transitions, in series or in parallel, and elimination of place or transitions' self-loops. The great advantage of these techniques is not only the preservation of the nets' properties after their application, but also the bidirectionality of that preservation. I.e., if the original net has the properties of safeness, boundness or liveness, then the reduced net preserves those properties, but also, if some of these properties are verified by the reduced net, then it is guarantied that the original net also verifies the same properties. Therefore, by applying the analysis techniques to the reduced net, we obtain the same results as if the original net was used at a lower processing cost.

There are several more elaborated approaches to reduction methods. In [Valette, 1979], a method is presented, which allows to analyse the properties of liveness and safeness of a Petri net in which one of its transitions has been replaced by a macro-transition. In [Suzuki and Murata, 1983] this method is generalized, and some ways of refining places of a Petri net using macro-places are proposed.

2.5 Petri net extensions and high-level Petri nets

Although the Petri net meta-model presented in the previous chapter – *place-transition Petri nets*, or just *PT-nets* – is already an evolution of the first Petri nets – *condition-event Petri nets* (*CE-nets*) – it still belongs to the class of low-level Petri nets. The relative simplicity of PT-nets derives from the fact that they are almost a direct result of the application of the ideas presented by Carl Adam Petri in his PhD dis-

sertation [Petri, 1962] and have the important advantage of being the most adequate for an introduction to Petri nets. However, this simplicity also carries disadvantages with it. One of the limitations of this meta-model is the complexity of the Petri net models that result from its application to "real-world" systems, due to the large number of nodes needed. Another disadvantage consists of some limitations of the expressive power of the PT-nets meta-model in certain situations.

As these limitations began to arise, several proposals have been made by the Petri net researchers' community, in the form of changes to the PT-nets meta-model. Extensions to the PT-nets meta-model have been proposed to deal with some of the limitations, but the most important contribution to reduce the real world models' complexity appeared only under the form of what are known as high-level Petri nets.

2.5.1 Continuous and hybrid Petri nets

One kind of extension that dramatically changed the discussion on Petri nets consists of considering that the number of tokens on the places of a Petri net may be expressed as a real number, instead of a non-negative integer. This extension originates the *continuous Petri net* meta-model, which expands the field of application of Petri nets beyond its original boundary – the discrete event systems. If only some places have a real number expressing their numbers of tokens, we are in the presence of a *hybrid Petri net*, containing a *discrete part* and a *continuous part* [David, 2005].

2.5.2 Inhibitor and enabling arcs

These special arcs have been introduced in some extended Petri net meta-models as a way of supporting coordination between two or more almost independent processes. The coordination supported by these special arcs can be synchronization or prioritization.

Fig. 2.8 illustrates the use of an inhibitor arc (represented by the dashed line ending with a small circle) for the establishment of a priority of process A over process B when there is an attempt of both processes to simultaneously access a common resource. As shown in the example, an inhibitor arc connects a place to a transition. If a token is in that place, the transition cannot fire. The inhibitor arc in this example ensures that, if tokens are present simultaneously in p_2 and p_6, t_2 will not fire until t_5 removes the token

that is inhibiting it in p_6, thus avoiding the conflict on the simultaneous access to the shared common resource, modeled by the token in p_3.

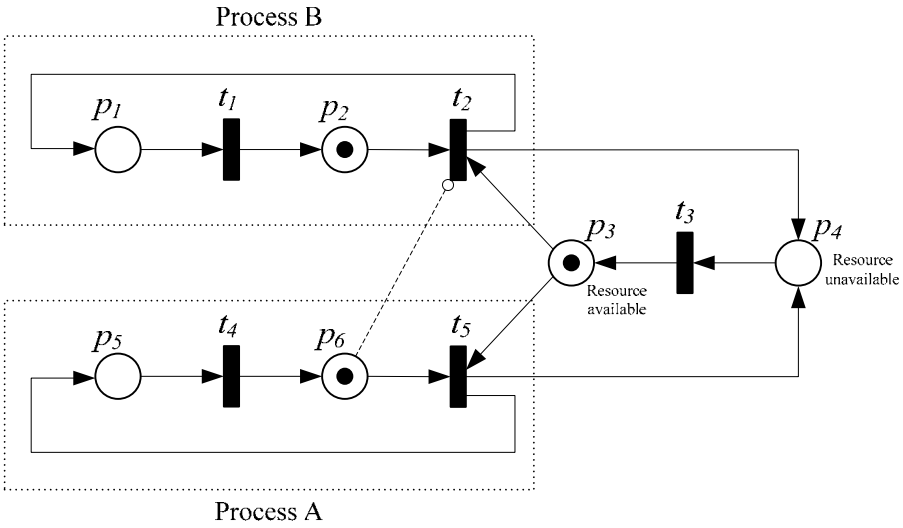

Fig. 2.8 – An inhibitor arc giving priority to process A over process B.

Fig. 2.9 illustrates the use of an enabling arc (represented by the dashed arrow) for the synchronization of process B with process A. An enabling arc connects a place to a transition, and it works as a normal arc in the sense that the transition to which it is connected will only fire if a token is present in the place where the arc has its origin. The difference from a normal arc is that when the transition fires, the token is not removed from the place. In the example of Fig. 2.9, the presence of a token in p_1 is not enough for t_1 to fire. Due to the enabling arc, t_1 has to wait for a token in p_3 so that it can fire.

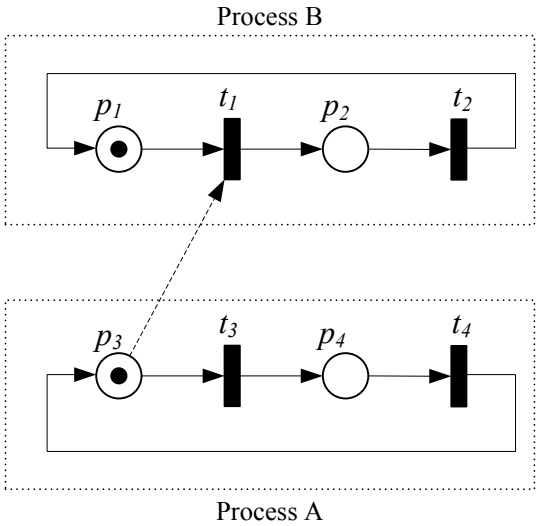

Fig. 2.9 – An enabling arc synchronizing process B with process A.

These are not the only special arcs that have been proposed to extend the PT-nets meta-model. For the version 2.0 of the *shobi-PN* meta-model, a *generalized set of arcs*, composed by 16 types of special arcs, is presented in [Machado and Fernandes, 2001], as one of two additional specifications to the version 1.0 of the *shobi-PN* meta-model [Machado and Fernandes, 1998].

2.5.3 Interpreted Petri nets

Interpreting a Petri net is to give a concrete meaning to a mathematical model, by associating existing entities to its places, transitions and tokens [Valette, 1993].

When considering a system that executes some kind of processing, the operations of this processing are associated to places or to transitions of a Petri net model. An operation that is associated to a place is considered to be executing during the whole period of time in which the place is marked. When associated to a transition it is supposed to be indivisible and executed instantaneously when the transition fires.

Some operations may not be executed immediately after the end of the preceding ones. Their execution may depend on the values of some data. Those situations are modeled through the association of *supplementary firing conditions* to the transitions that represent those operations. Therefore, in order for the firing of these transitions to occur they need to be enabled but, additionally, the associated supplementary firing conditions must be true. The supplementary firing conditions are often found in two or more transitions that constitute the postset of the same place, making possible to decide which transition can fire in a given moment, thus avoiding the inconveniencies of a situation that was described, in the previous chapter, as a structural conflict.

Synchronized Petri nets

In the case of *open systems* (systems that interact with their environment), some of the supplementary firing conditions may be associated to *external events*. This way, because these transitions must wait for those external events to occur, these Petri nets are called *synchronized Petri nets* and are often used to model, not a whole system, but only a controller for part of it. In these cases, the modeler must decide what part of the systems' events must be controlled by the controller and what part will be considered external to the controller. The occurrence of these external events will not be decided by the controller, but they may be considered important as inputs for the controller, in the form of external events for synchronization purposes.

Autonomous versus non-autonomous

Petri nets that integrate characteristics that connect them to their environment are also classified as *non-autonomous Petri nets*, because their execution depends on external data. In opposition, Petri nets whose execution does not depend on external data and are called *autonomous Petri nets*.

Control and data

Modeling an open system with a Petri net may be considered as structuring it in two parts. The first part, called the *control part* (or just *control*) is described by the uninterpreted Petri net and describes all potential sequences of events and activities. The second part, called the *data part* (or *operative part*, or *controlled part*) describes the internal data structures of the system, as well as the transformations and calculations that are operated on those data, without specifying in which instants they will take place. In addition to the internal data, these calculations may also involve the time and external data (information from the environment).

Considerations on the analysis of interpreted Petri nets

An interpreted Petri net can be seen as a PT-net to which a data layer has been added, by the addition of conditions that make the firing of enabled transitions dependent on the value of data. Although the majority of the Petri net models that are built to solve "real-world" problems are interpreted, this dependency of the data for the firing of transitions reduces the number of reachable markings of the uninterpreted Petri net (the original PT-net to which the data layer was added). Some analysis techniques (like place invariants) are still valid for interpreted Petri nets. However, the addition of the data layer makes the determination of the properties an *undecidable* problem, as complex as the formal proof of a computer program [Valette, 1993]. This means that the proof of some properties of the uninterpreted Petri net cannot be used to prove the same properties for the interpreted Petri net, but a lot of time may be saved in the modeling of a system if some problems have been previously found in the uninterpreted Petri net.

Structuring the data part

From what has been said here, it becomes apparent that interpreted Petri nets do not really constitute a new meta-model built from the original PT-nets. The basis for this statement consists of the fact that no rules or constructs have been defined for structuring the data part. For that reason, several extensions to support the operative part, in-

cluding timing aspects, have been proposed to Petri nets, originating new meta-models. Some of these extensions have also added to the basic PT-net meta-model a gain in expressive power, which allowed not only the construction of more compact models, but also modeling more complex systems.

Timed and Stochastic Petri nets

Timed Petri nets (with constant timings), or *stochastic Petri nets* (with stochastic timings with exponential distribution), are often used for performance evaluation of systems because they allow to perform that evaluation by analytical methods [David, 2005]. Time may be associated to places (in *P-timed Petri nets*) or to transitions (in *T-timed Petri nets*), being possible to transform one model into the other.

2.5.4 High-level Petri nets

The purpose of this topic is to explain why the CP-net meta-model was developed as well as what is its relation to Petri nets in general, and to high-level Petri nets in particular. That explanation is presented in the bibliographical remarks at the end of the first chapter, of volume 1, of [Jensen, 1997], given by the very creator of CP-nets, with a great clarity about the motivations for their development. That is why we chose to use that explanation here, in a summarized version, for most of this topic.

The emergence of several extensions originated a great diversity of Petri net meta-models, each one with particular strengths for coping with certain types of modeling problems. However, most of those net meta-models had almost no analytic power. Besides, it often turned out to be a difficult task to translate an analysis method developed for one net model into another.

The first meta-model of high-level Petri nets – Predicate/Transition Nets (PrT-nets) – designed to be of general purpose applicability, was presented in 1981 by Genrich and Lautenbach (and improved by Gendrich in 1987), forming a “nice” generalization of PT-nets and CE-nets. Because PrT-nets can be formally related to PT-nets and CE-nets, it is possible to generalize most of the basic concepts and analysis methods of these models, so that they also become applicable to PrT-nets. However, the generalization of the analysis methods of invariants, for application to PrT-nets, presented some technical problems, because the variables that appear in the arc expressions also appear in the invariants, making their interpretation prone to errors.

The first version of Colored Petri Nets (CP^{81}-nets) was defined by Jensen, in 1981, with the intent of overcoming that problem. The main ideas of this Petri net meta-model are directly inspired by PrT-nets, but they use functions attached to arcs (instead of expressions) that eliminate the presence of variables in invariants, making their interpretation non-problematic. However, the functions attached to arcs in CP^{81}-nets are often more difficult to interpret than the expressions attached to arcs in PrT-nets. This fact, and the strong relation between the two meta-models, inspired Jensen to propose, in 1985, an improved meta-model, named High-level Petri Nets (HL-nets), combining the qualities of PrT-nets and CP^{81}-nets. But because that name began to be used as a general classification for PrT-nets, CP^{81}-nets and HL-nets, Jensen decided to change the name of HL-nets to Colored Petri Nets (CP-nets), to avoid confusion.

CP-nets have two different representations, with formal translations defined between both, in both directions. The expression representation is nearly identical to PrT-nets (in their form presented in 1981), and was used to describe systems, while the function representation is nearly identical to CP^{81}-nets, and was used for all the different kinds of analysis. Nevertheless, because the linear function representation has turned out to be only necessary for invariant analysis, at present the expression representation is used for all purposes, except for the calculation of invariants.

Jensen refers that several other classes of high-level Petri nets have been defined, which are quite similar to CP-nets, but use different inscription languages (e.g., algebraic nets [Dimitrovici *et al.*, 1991], CP-nets with algebraic specifications [Vautherin, 1987], many-sorted high-level nets [Billington, 1989], numerical Petri nets [Billington, 1988], [Symons, 1978], OBJSA nets [Battiston *et al.*, 1988], PrE-nets with algebraic specifications [Kramer and Schmidt, 1987], Petri nets with structured tokens [Reisig, 1991], and relation nets [Reisig, 1985b]).

In [Gerogiannis *et al.*, 1998], a comparative presentation, evaluation and categorization of Petri nets meta-models is made, where hierarchical high-level Petri nets have been classified as having a “very high” level in the criteria: *descriptive power*, *degree of difficulty in mastering the method*, *compactness*, *degree of supporting encapsulation-abstraction-refinement* and *degree of specifying communication*, as opposed to low-level Petri nets, which have obtained a “low” level in most of those criteria, except for the last one, where their obtained level was “medium”. In contrast, for the criterion *ease of analysis*, the situation was the opposite, i.e., low-level Petri nets obtained a level

of “very high”, while hierarchical high-level Petri nets were assigned a level of only “medium”.

2.6 Conclusions

Petri nets are a graphical formalism with an underlying strong mathematical foundation that can be applied in systems specification, analysis and verification. They represent systems as a structured set of interconnected active and passive elements – transitions and places connected by arcs. They are also executable through a mechanism called firing of transitions, which removes tokens from some places and adds tokens to other places. A system state is represented by a particular distribution of tokens. Petri nets are particularly adequate to describe and analyze systems that are characterized as asynchronous, distributed, parallel and nondeterministic.

Chapter 3

CP-nets for Animation Prototypes

This chapter presents a proposal of a technique for modeling high-level user requirements of the functionalities of workflows (i.e., the business processes to be automated by those workflows) in UML use case diagrams and some kind of stereotyped sequence diagrams, and, by applying simple transformation rules, deriving CP-nets from those UML models. The simulation of those CP-nets will control animation prototypes that will be used to submit the modeled requirements to stakeholders' validation.

3.1 Requirements validation at early stages

Requirements elicitation is all about learning and understanding the needs of users and project sponsors with the ultimate aim of communicating these needs to the system developers [Zowghi and Coulin, 2005]. Getting the right requirements is considered as a vital and difficult part of software development projects. Modeling and model-driven approaches provide ways of representing the existing or future processes and systems using analytical techniques with the intention of investigating their characteristics and limits [Machado *et al.*, 2005a].

The validation of functional user requirements is a cyclic process, as depicted in Fig. 3.1. The first step of that cycle consists of the elicitation of the requirements, as shown in Fig. 3.1a. On the second step, a model of the system's functionalities is built, according to the elicited requirements. Finally, on the third step, the model is submitted to the appreciation of the stakeholders (users) in order to validate it. But, most likely, this first model does not receive a complete validation, implying a redefinition of the

requirements, originating a new cycle, as represented in Fig. 3.1b. This cycle may need to be repeated more times, until a complete validation of the model is achieved.

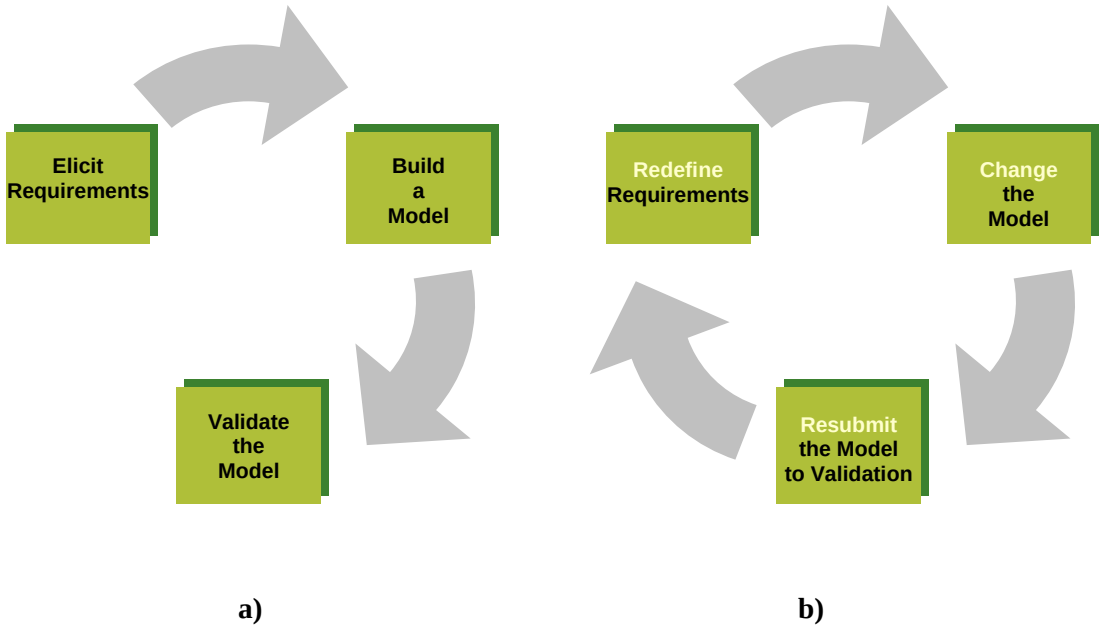

Fig. 3.1 – The requirements validation cycle.

Naturally, the costs involved in each cycle are proportional to the complexity of the model. Furthermore, the stakeholders involved in the early stages of the requirements process are, typically, the persons with the highest rank in the organization's hierarchy, who are usually neither interested in detailed systems' descriptions (their interests are essentially confined to the business level), nor willing to spend more than just the strictly indispensable time with the requirements validation task. In order to reduce as much as possible these costs, the detail of the models used in these early stages of the requirements process must also be kept reduced to the minimum needed to satisfy the higher ranked stakeholders' interest.

3.2 High-level UML modeling

UML use case diagrams are a quite adequate tool to describe user requirements at a first high level of abstraction. These diagrams constitute a suitable means for delimiting the system boundaries, for identifying the functionalities that should be provided by the

system, and for affecting external actors with specific use case functionalities. Additionally, brief textual descriptions may be provided in natural language for each use case. These diagrams are normally constructed by the developers in an attempt to document the elicited requirements. With the support of those textual descriptions, stakeholders can read and use these diagrams to recognize the main functional areas of the system to be designed.

General functionalities of the uPAIN system are inscribed in the UML use case diagram depicted in Fig. 3.2. A set of additional use case diagrams, as the one of Fig. 3.3, have been constructed to refine some of the use cases existent in Fig. 3.2. The corresponding textual descriptions have also been obtained.

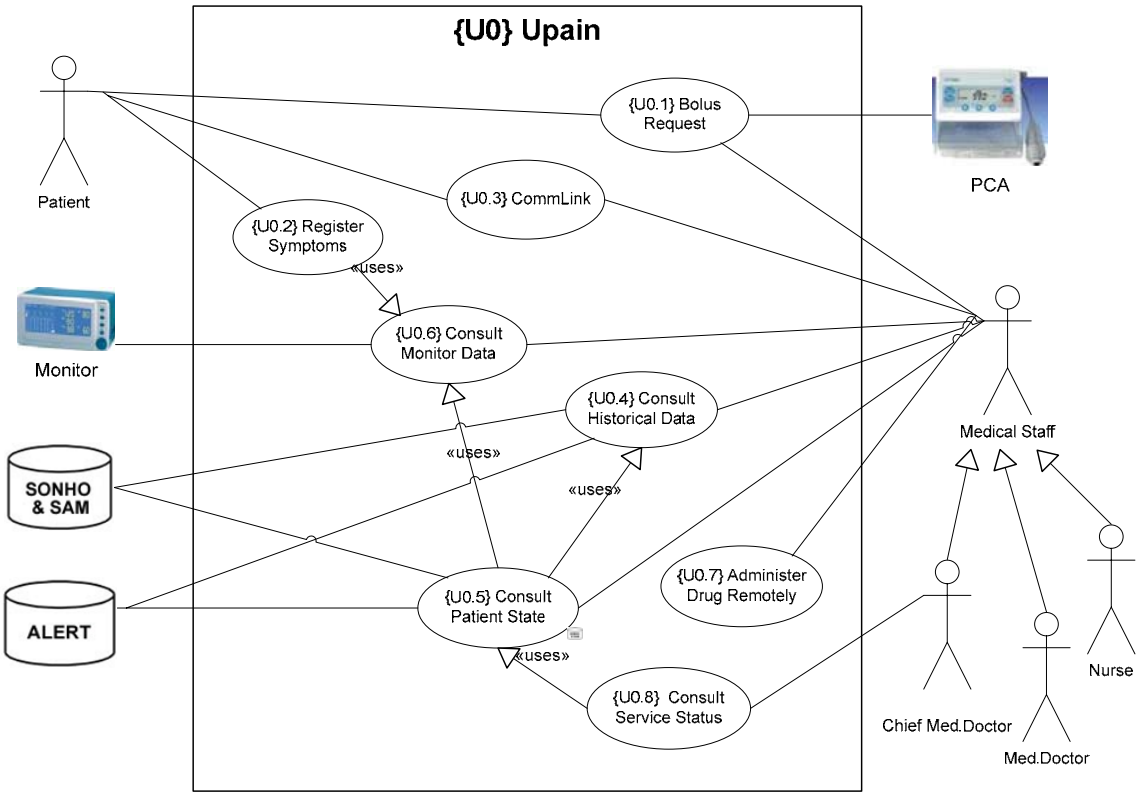

Fig. 3.2 – UML use case diagram for the uPAIN system.

With the exception of just a few *«uses»* and *«extends»* relationships that may already be shown between use cases, it turns out to be obvious that use case diagrams do not practically say anything about how the system should be designed, in order to supply the identified functionalities. A further step on that direction may be provided by sequence diagrams in order to illustrate the desired dynamic behavior in what concerns its

functional interaction with the environment. These diagrams are also to be constructed by the developers. Stakeholders can also read them. However, stakeholders are not comfortable with all the details these diagrams can entail.

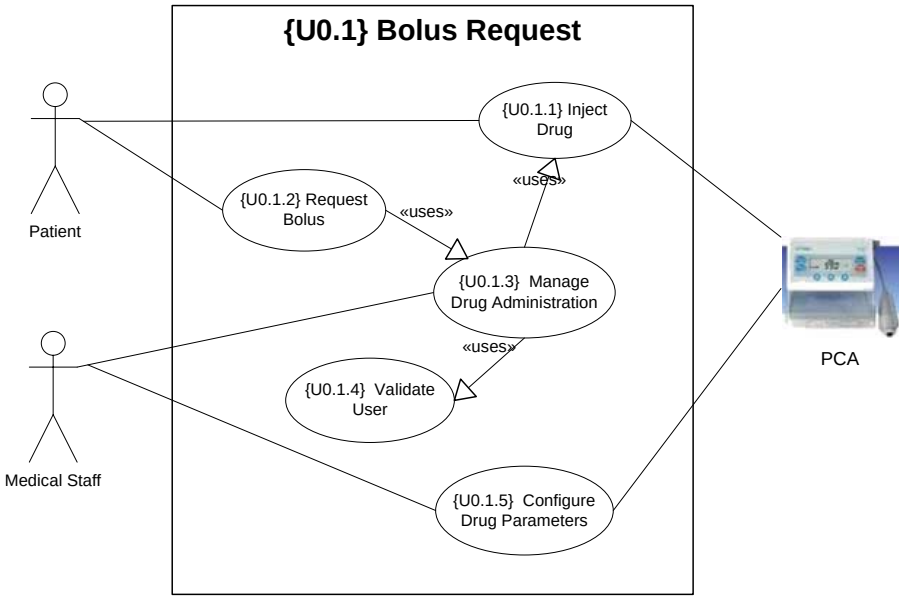

Fig. 3.3 – UML case diagram detailing the use case *{U0.1} Bolus Request*.

3.2.1 Stereotyped sequence diagrams

At the analysis phase of system development, we adopt a stereotyped version of UML sequence diagrams, where only actors and use cases are involved in the sequences, since no particular structural elements of the systems are known yet. This is illustrated by the sequence diagram of Fig. 3.4, whose purpose is to model the exchange of messages among the external actors and use cases depicted in Fig. 3.3, thus representing just a small increase in semantics detail to the use case diagram. Sequence diagrams of this kind allow a pure functional representation of behavioral interaction with the environment and are particularly appropriate to illustrate workflow user requirements. Additionally, these stereotyped sequence diagrams keep the model detail at the higher level, making them the most adequate for the early stages of user requirements modeling.

The main aspect that makes our stereotyped UML sequence diagrams contrast with the traditional ones consists of the fact that these already involve system objects in the interaction with external actors, implying that those objects must be previously identified. One important issue concerning objects identification and building object diagrams is that they already model structural elements of the system, which is clearly beyond the

scope of the user requirements. Additionally, the use of this kind of traditional sequence diagrams at the first stage of the analysis phase (user requirements modeling and validation) requires a deeper intervention of modeling skills that are hardly understandable to most stakeholders, making more difficult for them to establish a direct correspondence between what they initially stated as functional requirements and what the model already describes. Therefore, a validation of the user requirements resulting from such an advanced model is not only more difficult to achieve, but also less trustworthy and less ensuring that the resulting system will correspond effectively to the stakeholders expectations.

Fig. 3.4 depicts one stereotyped UML sequence diagram for the uPAIN system that describes one macro scenario where a patient requests a bolus. That request may be accepted by the system or originate a request for an explicit medical decision. In the later case, the doctor may decide to authorize the bolus or to reconfigure the PCA parameters.

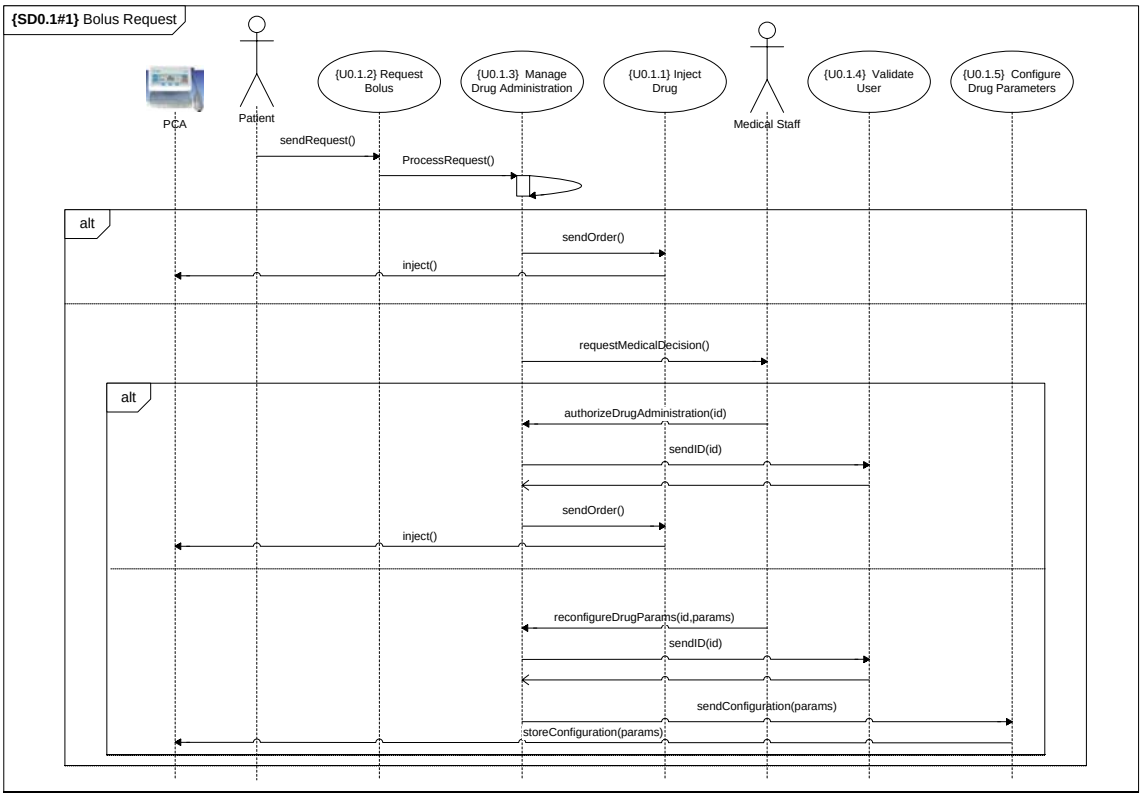

Fig. 3.4 – UML sequence diagram of a macro-scenario for the uPAIN system.

The integration of several scenarios into only on sequence diagram (for a macro scenario) is possible due to the new mechanisms of UML 2.0 (*alt-blocks*, in this case) in

supporting different kinds of frames. As can be seen in the diagram, this first step is decided by the use case *{U01.3} Manage Drug Administration*. The first frame of the macro scenario (accept bolus) corresponds to the first section of the outer alt-block. The second section of the outer alt-block corresponds to the scenario in which the use case *{U01.3} Manage Drug Administration* decides to request a medical decision. The second and third frames correspond to the two sections of the inner alt-block, where the doctor may decide to authorize the bolus or to reconfigure the PCA parameters.

Some more stereotyped UML sequence diagrams have been constructed to capture the main system scenarios.

3.3 Colored Petri Nets

In the previous chapter, the power of low-level Petri nets was presented and explained in its several aspects. The most important limitations of low-level Petri nets and some of their most important extensions, as well as more powerful Petri net languages (high-level Petri nets) were also discussed. Since Colored Petri Nets (CP-nets) and CPN-Tools were used in the uPAIN demonstration case, a brief discussion and a definition of CP-nets are presented here.

Colors are the most important constituent of the CP-net meta-model, in the sense that they represent the groundwork for all the other elements that distinguish CP-nets from low-level Petri nets. A color is a value that can be *bound* (associated) to a token. This association of values to tokens increases dramatically the expressive power of Petri nets. Arc expressions use variables that can also be bound to colors, making the evaluation of the expressions dependant of colors. The colors that are bound to tokens result either from the evaluation of the expressions of the output arcs of a transition, or from *initialization expressions* that are associated to places and evaluated in the beginning of the execution of the CP-net to provide an initial marking. Variables are also used in boolean expressions, called *guards*, that are associated to transitions and make the occurrence (or firing) of transitions dependent on the colors that are bound to the guards' variables. Furthermore, the occurrence of a transition depends not only on the presence of a certain number of tokens in its input places, but also on the colors that are bound to those tokens.

3.3.1 Structure

The definition of a non-hierarchical CP-net, which can be found in [Jensen, 1997], is as follows:

A *non-hierarchical colored Petri net* is a tuple

$CPN = (\Sigma, P, T, A, N, C, G, E, I)$

that satisfies the following requirements:

(i) Σ is a finite set of non-empty types, called *color sets*.

(ii) P is a finite set of places.

(iii) T is a finite set of transitions.

(iv) A is a finite set of arcs such that: $P \cap T = P \cap A = T \cap A = \emptyset$.

(v) N is a *node* function: $N: A \rightarrow P \times T \cup T \times P$.

(vi) C is a *color* function: $C: P \rightarrow \Sigma$.

(vii) G is a *guard* function: $G: T \rightarrow \{expressions\}$ such that:

$\forall\, t \in T: [Type(G(t)) = \{true, false\} \wedge Type(Var(G(t))) \subseteq \Sigma]$.

(viii) E is an arc expression function: $E: A \rightarrow \{expressions\}$ such that:

$\forall\, a \in A: [Type(Var(E(a))) \subseteq \Sigma \wedge Type(E(a)) = C(p(a))_{MS}]$

where $p(a)$ is the place of $N(a)$.

(ix) I is an initialization function: $I: P \rightarrow \{closed\ expressions\}$ such that:

$\forall\, p \in P: [Type(I(p)) = C(p)_{MS}]$.

The interpretation of this definition is as follows:

(i) *The set Σ of color sets*

The set Σ of color sets determines the types, operations and functions that can be used in the *net inscriptions* (arc expressions, guards, initialization expressions, etc.).

(ii), (iii) and (iv) *The sets P, T and A*

As in low-level Petri nets, the places, transitions and arcs distinguish from each other because they are entities with different semantics and characteristics. Therefore, the sets P, T and A are required to be mutually disjoint, as expressed in (iv). However, this definition does not forbid the net structure to be empty (i.e., $P \cup T \cup A = \emptyset$ is allowed). This

allows a modeler to write the declarations for a CP-net (the definition of the set of color sets for that CP-net) and submit them to a syntax check, with the CPN-Tools, before beginning to draw the net structure.

(v) *The node function N*

By mapping arcs into pairs of nodes, the node function N tells us what are the source and destination nodes of each arc. For instance, $N(a) = (p_3, t_2)$ means that the arc a has the place p_3 as its source node and the transition t_2 as its destination node.

In CP-nets the existence of more than one arc having the same input and output nodes is allowed. This is why the set of arcs is defined by means of a specific set A, together with a mapping of its elements into the elements of $P \times T \cup T \times P$, and not as a mere subset of $P \times T \cup T \times P$, as is the case with low-level Petri nets.

It is also allowed for a node to be isolated, i.e., having a node that is not connected by arcs to any other node of the CP-net. This situation may occur intentionally, when the modeler inserts an isolated node in the model, or unintentionally, when, for a given binding, the arc expressions of all the input and output arcs of a node evaluate to the empty *multi-set* (a multi-set may be seen as a set that accepts multiple occurrences of the same element). If an arc expression always evaluates to the empty multi-set, that is equivalent to the inexistence of the correspondent arc, (this is the equivalent situation of an arc with a weight of zero, in low-level Petri nets).

(vi) *The color function C*

The mapping of each place to a color set, given by the color function C, represents the obligation of every token color in a given place p to belong to the same color set. Therefore, $C(p)$ represents the color set associated to the place p.

(vii), (viii) and (ix) *Type*(*argument*)

The expression *Type*(*argument*) is used to designate the color set to which the value of *argument* belongs.

(vii) and (viii) *Var*(*argument*)

The expression *Var*(*argument*) designates the set of all the variables used in *argument*.

(vii) *The guard function G*

The mapping of each transition to a boolean expression, given by the guard function G in (vii), establishes that each transition t must have a guard $G(t)$ conditioning its occurrence. In order to ensure consistency, the CPN-Tools assume that the closed expression *true* is the guard associated to every transition to which the modeler did not explicitly add an expression as its guard. $Type(G(t)) = \{true, false\}$ means that the type of a guard must be boolean, and $Type(Var(G(t))) \subseteq \Sigma$ means that all the variables used in the guard must be of types belonging to the color sets of the CP-net.

(viii) *The arc expression function E*

The arc expression function maps each arc a to an expression $E(a)$ (an *arc expression*), i.e., each arc must have an expression attached to it. Additionally, this expression must satisfy two conditions: (1) its variables must be of types belonging to the set of color sets declared for the net; and (2) its type must be $C(p(a))_{MS}$, i.e., the arc expression must evaluate to a multi-set over the color set that is attached to the place to which the arc is connected. This second condition contains two very important features:

a) The first feature obligates all of the arcs connected to a given place to have expressions that generate values of the same type of the tokens that are allowed into that place. The colors (or values) of the tokens that are put into a place are generated by the evaluation of the expressions of its input arcs. In order to be capable of removing tokens from a place, the expressions of its output arcs must evaluate to colors of the same type of the colors of the tokens.

b) The second feature is the need for the type of the arc expressions to be multi-sets. It may be necessary for one arc to move more than one to-

ken at a time. For that purpose, its expression must be capable of generating multiple colors in one evaluation. This is only possible if the expression evaluates to a multi-set. For instance, if we want an arc expression to generate two tokens at a time, it must be of the form

2`(*expr*)

However, when an expression is required to generate only one value, the CPN-Tools allow the simplified form

expr

instead of

1`(*expr*)

(ix) *The initialization function I*

The initialization function maps each place p into a closed expression of type $C(p)_{MS}$, i.e., a multi-set over the color set of the place p. A closed expression is an expression that does not contain variables and, consequently, it always evaluates to the same color, for every possible binding. Its purpose is to provide an initial state for the CP-net, i.e., an initial marking. Because a state does not require tokens in every place of the net, the initialization expression is not mandatory. Analogously to (viii) it may have the form

expr

instead of

1`(expr)

when only one token is needed for a given place.

3.3.2 Behavior

With the exception of the additional condition imposed by guards to the firing of transitions, the behavior of CP-nets follows, basically, the same rules as for PT-nets. Each input place of a transition has a number of tokens that is expressed as a multi-set over the color set that is associated to the place. The number of elements of that multi-set is the number of tokens in the place. As with PT-nets, the number of tokens

that a transition needs in each input place, to become enabled, must be greater or equal to the number of tokens that is determined by the evaluation of the arc expressions of the corresponding input arcs.

However, because the evaluation of arc expressions depends on bindings, the state of enablement of a transition depends also on bindings. Additionally, for a transition to be considered enabled its guard must also evaluate to *true* for the same binding that is used to evaluate the arc expressions. Therefore, it is necessary to specify the binding *b* for which a transition *t* is enabled, in a given marking *M*. That is expressed:

t is enabled in *M* for the binding *b*

or

(*t*,*b*) is enabled in *M*

where (*t*,*b*) is what is called a *binding element*.

When a transition *t* fires in the binding *b* we say that (*t*,*b*) *occurs*.

3.3.3 Dynamic character of the structure

It has already been said above (in (v), the explanation of the node function) that an arc expression may evaluate to the empty multi-set. It was also said that, if an expression evaluates to the empty multi-set for every possible binding, that situation has an equivalent in low-level Petri nets that consists of considering that a non-existent arc is an arc with a weight of zero. However, this similarity exists only in a case that is considered a design abnormality in a CP-net (an arc expression that always evaluates to the same multi-set, i.e., an arc expression that is a closed expression).

This discussion raises a very important difference between these two types of Petri nets. While, in low-level Petri nets, arc expressions are constants (integer weights), determining a static structure, in CP-nets arc expressions use variables and, consequently, they normally evaluate to different multi-sets, depending on the bindings that take place during the net's execution. This non-deterministic variability, which is naturally induced by the diversity of bindings that take place during the CP-net execution, may also be explicitly imposed by the modeler, by the use of *if-then-else* or *case* structures in arc expressions. Therefore, during the execution of a CP-net everything works as if its structure was dynamic. Depending on the evaluation of the net's arc expressions, each arc may exist (i.e., if it does not evaluate to the empty multi-set) or not. Guards have

also a very important role in this dynamic character of the structure of CP-nets. When a guard evaluates to *false* the CP-net behaves as if the corresponding transition, along with all its surrounding arcs, did not exist.

3.3.4 Code segments

Code segments are pieces of sequential code that may be associated to transitions. The code segment of a given transition is executed each time a binding element of that transition occurs. Code segments may contain an *input* pattern, an *output* pattern, a *code guard* and an *action* body. The action body contains the executable code. The input pattern is used to pass the values of some of the variables of the occurring binding element to the executable code. This allows the code segment to use, but not change, the values of data from the net during its execution. However, by the use of an output pattern it is possible to change the values of some variables in the code segment, as long as those variables do not appear in the input arcs' expressions or in the guard (because these influence the transition's enabling).

Code segments are an extremely powerful instrument. Through them, a CP-net may be used to control, during its execution, the launching of virtually any action that is usually external to a CP-net's execution. Among many other things, they may be used to write data to output files, read from input files, or, as was done in our demonstration case, execute graphical animations related to the modeled system. Examples of code segments may be seen in several of the transitions of figures 3.11 and 3.12.

3.3.5 Equivalent PT-nets

It can be demonstrated [Jensen, 1997] that any PT-net may be represented by a CP-net. Conversely, it can be demonstrated that the reverse transformation is also always possible, i.e., any CP-net can be represented by an equivalent PT-net. The most important aspect of these equivalences is that all the properties that are verified by a PT-net are also verified by its equivalent CP-net, and vice-versa. This does not imply, however, that it is necessary to obtain the equivalent PT-net, of a given CP-net, to check the properties of the later, because all the analysis methods are directly applied to the CP-nets.

3.3.6 Hierarchy

As was said in the previous chapter, hierarchy is the approach by which Petri nets may be structured, to allow an incremental model design, weather it is by a top-down, bottom-up, or mixed strategy, in a very similar way as procedures and functions do for structured programming.

It was said above that any non-hierarchical CP-net can be translated in a behaviorally equivalent PT-net, and vice-versa, and that this allows the use, with CP-nets, of the same analysis techniques that are used for PT-nets. In a similar way, CP-nets implement hierarchy by means of two formal constructs – *substitution transitions* and *fusion places* – providing a well-defined semantics that allows to demonstrate that any hierarchical CP-net can be translated into a behaviorally equivalent non-hierarchical CP-net, and vice-versa, thus making it possible to apply to hierarchical CP-nets the same analysis techniques that are used for non-hierarchical CP-nets.

In informal terms, a substitution transition may be defined as a special type of transition that has a corresponding subnet describing the detail of the action represented by that transition. It may have begun by being an ordinary transition in a single non-hierarchical CP-net, which models a first level of detail of a system, receiving tokens from its input places and placing tokens in its output places.

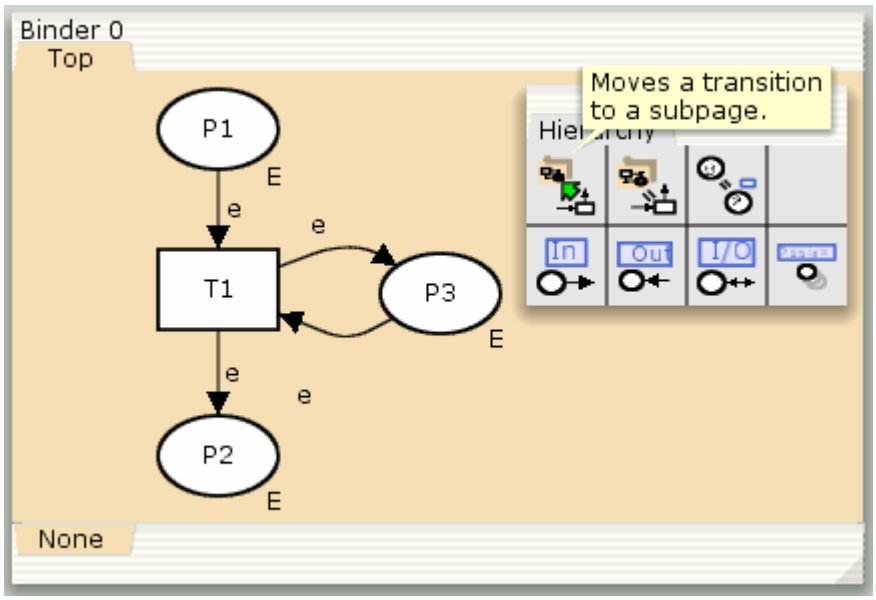

Fig. 3.5 – An example CP-net before applying the *MoveToSubPage* command to T1.

When the modeler decides to model the detail of the action that a transition represents at the first level, in CPN-Tools he simply applies the *MoveToSubPage* command (see Fig. 3.5) to the transition. As a result, a new *page* is created, containing a copy of the transition, along with all its surrounding arcs and places. Fig. 3.5 shows an example of a CP-net, before applying the *MoveToSubPage* command to the transition T1. Fig. 3.6a shows the changes that occurred in the CP-net of Fig. 3.5 after applying the *MoveToSubPage* command to T1, and Fig. 3.6b shows the new subpage that was automatically created by the *MoveToSubPage* command.

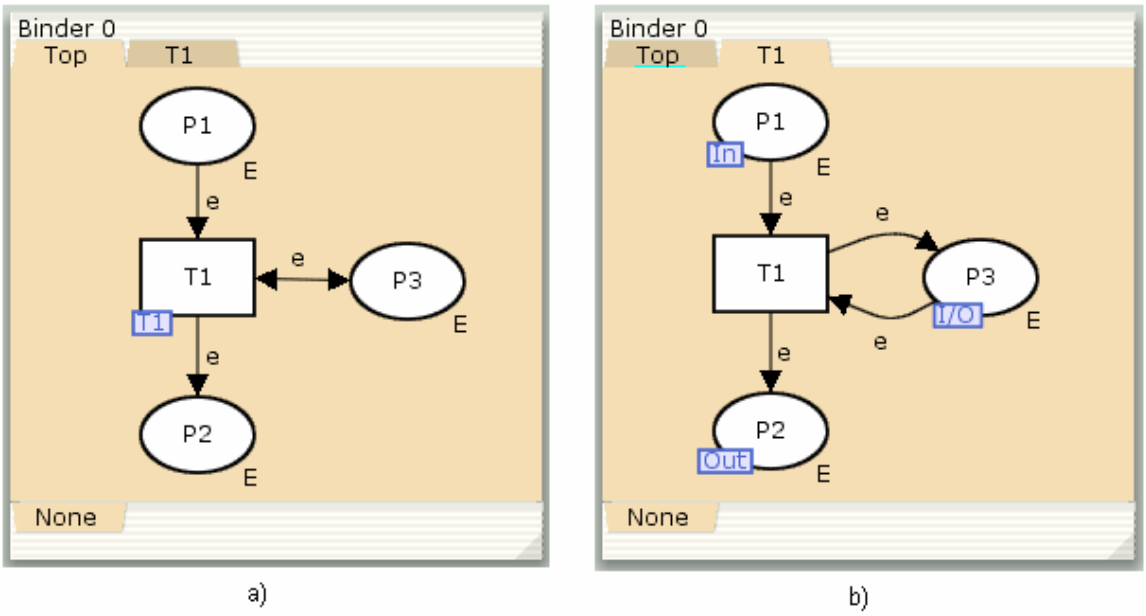

Fig. 3.6 – The CP-net of Fig.3.5 after applying the *MoveToSubPage* command to T1.

The original non-hierarchical CP-net has now become a hierarchical CP-net composed by two *pages*. The originally unique page (the one named "Top") has now become a *superpage*, because it contains a *supernode* (the substitution transition T1). The newly created page is a *subpage*, because it contains the detail of a supernode. After applying the *MoveToSubPage* command to T1, it is transformed into a substitution transition (see Fig. 3.6a). That is revealed by the new tag attached to it, containing the name of the corresponding subpage (it is automatically named with the same name of the original transition).

The tags *In*, *Out* and *I/O* that appear on the places P1, P2 and P3, respectively, of the subpage "T1", indicate that these places are, respectively, an *input port*, an *output port* and an *input/output port* of the subpage. They are also *fusion places*, i.e., the places P1,

of the page “Top”, and P1, of the subpage “T1” constitute a *fusion set*, which means that they are two appearances, of the same place, in different pages of the hierarchical CP-net (the places named P2, of both pages, make up another fusion set, and the ones named P3 make up a third fusion set). In practical terms, it means that any token that is put into one of the places of a fusion set also appears in all of the places that belong to the same fusion set. Likewise, any token that is removed from of one of the places of a fusion set also disappears from all of the places that belong to the same fusion set. The places of a superpage that correspond to the ports of a subpage are called *sockets*.

The net of the newly created subpage may then be changed to reflect the detailed action of the substitution transition (see Fig. 3.7b). The subpage may also be renamed, originating the same automatic renaming of the tag of the corresponding substitution transition (see Fig. 3.7a). If considered convenient, after the creation of a subpage the places that constitute its ports may be renamed at will (as was done in the CP-net of Fig. 3.10), because CPN-Tools maintain internal identifiers for each node, and it is those identifiers that are used to assure the connection between each port and its corresponding socket.

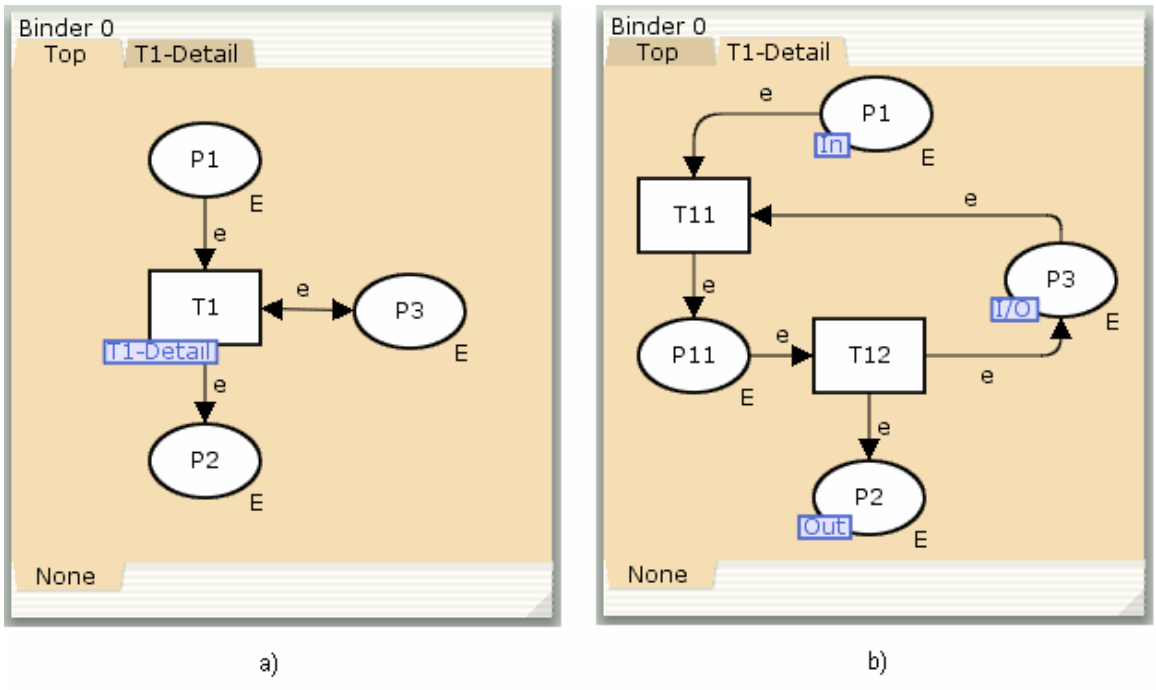

Fig. 3.7 – The CP-net of Fig.3.6b after a possible modeling of the detail of T1.

In order to keep this example simple, the CP-net of the page “Top” contains just the minimum set of nodes necessary to illustrate the process of transforming it into a

hierarchical CP-net. It does not contain any other node except the substitution transition and its input and output places. For that reason, if the modeler decided to refine the transition T1 directly in the original CP-net, instead of creating a subpage, he would obtain a non-hierarchical CP-net exactly coincident with the one of Fig. 3.7b. However, in a real modeling situation, the CP-net of a superpage contains other nodes that are not transferred to a subpage when a transition is moved to it for detailing purposes. That can be seen in the examples of the Figures 3.11 and 3.12. The substitution transition "Bolus Request" of Fig. 3.11 is detailed in a subpage containing the CP-net of Fig. 3.10. The only input place of the substitution transition "Bolus Request", called "Patient Option Chosen" is fused to the only input port of the CP-net of Fig. 3.10 – the place named "Strong Pain". Similarly, the only output place of the substitution transition "Bolus Request", called "Start End" is fused to the only output port of the CP-net of Fig. 3.10 – the place named "Done". None of the other nodes of the CP-net of Fig. 3.11 is represented in the "Bolus Request" subpage. In this case, if the modeler decided to refine the transition "Bolus Request" directly in the CP-net of Fig. 3.11, the resulting CP-net would be much larger and more difficult to read than the hierarchical CP-net.

Formal definitions for hierarchical CP-nets and their behavior are also presented in [Jensen, 1997], but because the construction of hierarchical CP-nets from non-hierarchical CP-nets is easily understandable by means of plain informal descriptions and explanations (contrarily to what happens with the introduction of non-hierarchical CP-nets, departing from PT-nets), we decided not to present them here.

3.4 CP-nets for Animation Prototypes

The behavior of the animation prototypes (proposed in this dissertation) results from translations of sequence diagrams into CP-nets. The transitions of these CP-nets present a strict one to one relationship with the messages in the sequence diagrams, i.e., for each message in a sequence diagram, one transition, in the corresponding CP-net, is created. In order to make that correspondence more evident, the name of each transition matches exactly the name of the corresponding message in the sequence diagram. Two simple rules were used for that translation: (1) Fig. 3.8 illustrates the rule for translating two successive messages in a sequence diagram (Fig. 3.8a) into a CP-net (Fig. 3.8b); (2) Fig. 3.9 illustrates the rule for translating an alternative block in a sequence diagram (Fig. 3.9a) into a CP-net (Fig. 3.9b).

With these transformation rules, the color set of all of the places is the color set *E* (or *unit*), i.e., all the tokens are the uncolored *e* token (equivalent to the tokens in PT-nets), as can be seen in Fig. 3.10. This is why the arc expressions were omitted in Figs. 3.9 and 3.10.

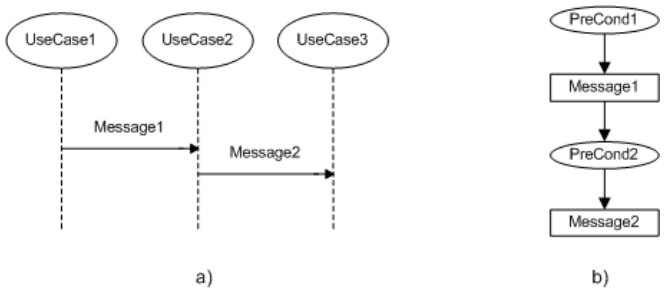

Fig. 3.8 – Transformation of successive messages.

Because all the tokens are *e* tokens, the evaluation of the expressions used in the conditions of the output arcs of a transition that has two alternative output places (e.g. transition *Message1* of Fig. 3.9b) cannot, obviously, depend on the color of the tokens. Instead, the expressions in those conditions use variables that are bound by the *Output* pattern of the code segment of that transition. That situation is illustrated by the transitions *Process Request* and *Request Medical Decision* of the example CP-net of Fig. 3.10.

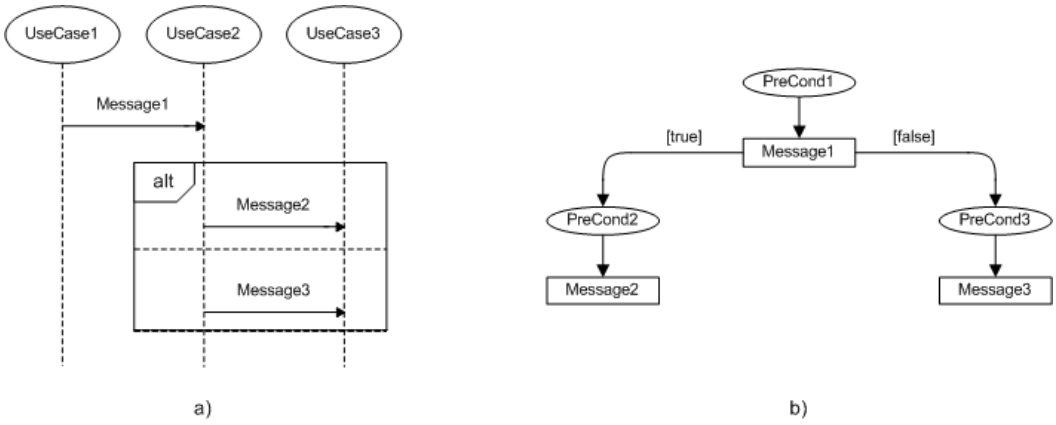

Fig. 3.9 – Transformation of an alternative block.

By just observing Figs. 3.9 and 3.10, it results clear that, with these transformation rules, a place of the CP-net corresponds to the interface between two consecutive messages of the sequence diagram. Therefore, a place represents a part of a use case of the stereotyped sequence diagrams, which responds to a precedent message with a subsequent message. If the refinement of a given use case is modeled with a new sequence

diagram, this new sequence diagram can, in turn, be transformed into a refinement subnet. Although it is not possible to create sub-pages for places (there are no such things as "substitution places"), those places may be replaced, at the same hierarchy level where they originally appear, by refinement subnets (composed of one input place, one output place, and one substitution transition between them) to support the refinement of use cases. This way, the refinement subpage of each substitution transition of such refinement subnets represents the refinement of part, or the totality, of a use case.

Typically, those refinement subnets will be built after the application of the 4SRS (4 step rule-set) technique [Machado *et al.*, 2005b] that transforms users requirements into architectural models representing system requirements, by mapping use cases into system level objects (or components) within a four step approach: (1) object creation, (2) object elimination, (3) object packaging and aggregation, and (4) object association. Therefore, each transition in those subnets will correspond to the invocation of a method of a system object.

The CP-net of Fig. 3.10 is responsible for the animation of the use case *{U0.1} Bolus Request* (see Fig. 3.3) by executing the scenarios described in Fig. 3.4, each one corresponding to one of the three branches of the CP-net. Those nodes and arcs drawn with thinner lines were added in a later phase, and have no semantic correspondence to the sequence diagram. They were included for the purpose of tools interoperability, as explained in chapter 4.

The CP-net represented in Fig. 3.11 corresponds to the top-level net of the animation prototype for the uPAIN system. Thick lines were used to represent the elements that correspond to the main animation paths. Most of the transitions of this CP-net correspond to the use cases in Fig. 3.2. For example, the refined CP-net of the substitution transition *bolus request* of Fig. 3.11 corresponds to Fig. 3.10.

Because the transformation rules depicted in Fig. 3.8 and Fig. 3.9 are to be applied only to sequence diagrams, direct links between the use case diagram of Fig. 3.2 and the CP-net of Fig. 3.11 were not intended to follow explicit rules. Instead, the link between the UML diagrams and the CP-nets is obtained in two steps: (1) sequence diagrams are constructed after identifying scenarios that involve use cases and actors; (2) CP-nets are derived from the sequence diagrams by applying the transformation rules.

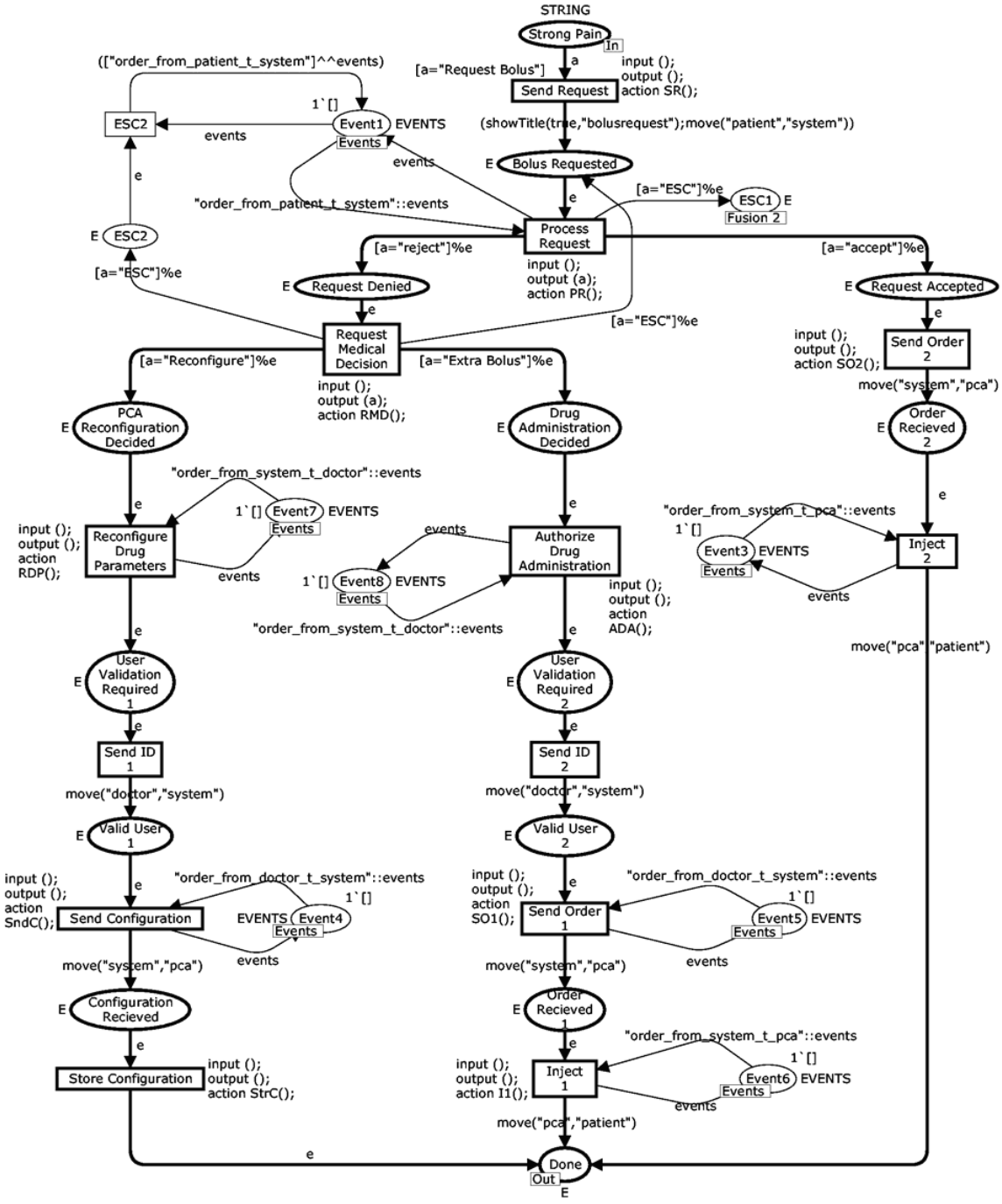

Fig. 3.10 – CP-net responsible for the animation of the use case *{U0.1} Bolus Request*.

Sequence diagrams transmit partial views for the interaction between the system and its environment, allowing the adoption of an evolutionary approach, by considering a set of sequence diagrams to have a partial evaluation of the requirements and then progress with more detailed requirements. In the uPAIN system, the animation prototype reflects only a top-level description of the system. After the validation of this top-level model, a set of additional animations, based on refined sequence diagrams at the solution level (where objects would already appear), can be constructed.

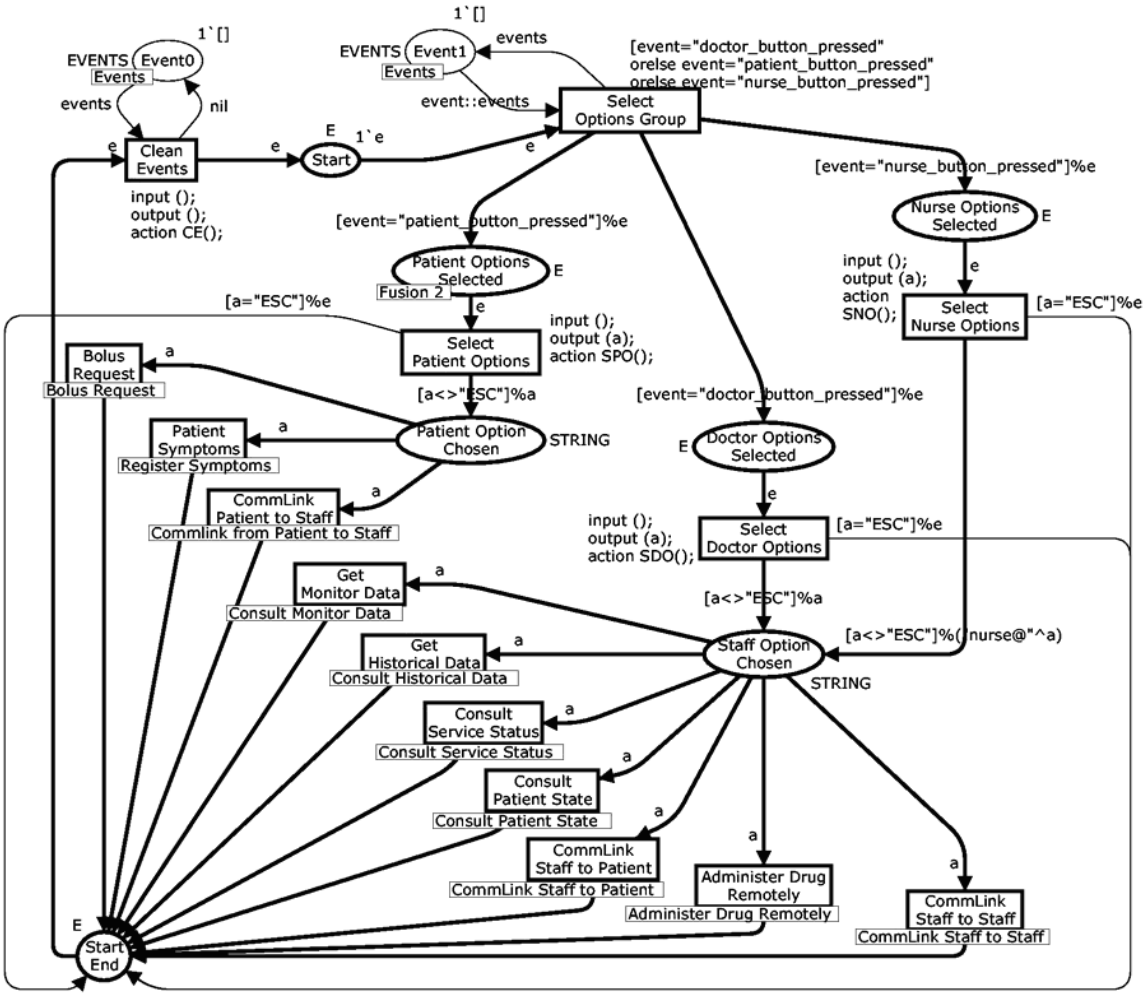

Fig. 3.11 – Top-level CP-net of the animation prototype for the uPAIN system.

3.5 Discussion and conclusions

In this chapter, a technique for deriving CP-nets from UML models representing high-level user requirements of the functionalities of workflows was proposed. To obtain those CP-nets, two transformation rules were suggested.

One advantage of the transformation rules suggested is that the resulting CP-nets are structurally simple and require only uncolored tokens on the workflow paths. This would not be the case, if for the transformation rule of an alternative block of a sequence diagram (see Fig. 3.9), a conflict place (a place with two output arcs) was used, as depicted in Fig. 3.12, instead of a transition with two output arcs, as shown in Fig. 3.9b.

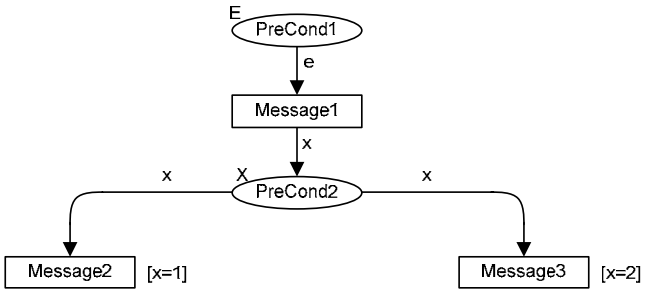

Fig. 3.12 – Using a conflict place for the transformation of an alternative block.

As shown in Fig. 3.12, if a conflict place was used to the transformation of an alternative block of a sequence diagram, the decision whether *Message2* or *Message3* should follow *Message1*, would have to be taken from the evaluation of guards associated to transitions *Message2* and *Message3*, depending on the value (color) of the token in place *PreCond2*. Therefore, the color set of place *PreCond2* would have to be other than *E*. For that reason, a color set *X* was used for the place *PreCond2*, and a variable *x*, of that color set, was used for the surrounding arcs' expressions. The variable *x* would have to be bound by the *Output* pattern of the code segment of the transition *Message1*.

At an initial phase of simulations where these CP-nets are to be used, only the control of animation prototypes, for validation of the workflows by the stakeholders, is intended. For that reason, the uncolored tokens in the workflow paths of the CP-nets is a plus, because in case of need of simulations of the CP-net model, of other types than the mere control of those animations (e.g. for performance analysis purposes), the change of the tokens' semantics, by means of the addition of colors, would be independent of the workflows' control logic, and, therefore, would not interfere with it.

Chapter 4

uPAIN Demonstration Case

As already said, the uPAIN demonstration case was used to experiment and evaluate an approach of user requirements validation by means of the execution of UML models. The main idea was, in a first step, to elicit the users' requirements and represent them in UML models. In a second step, these models were presented and explained to the stakeholders in order to subject the elicited requirements to a previous validation. This previous validation of the elicited requirements through the UML models served two purposes: (1) it reduced the number of cycles of reconstruction of animations before their complete final validation; (2) the observation of the stakeholders' reactions to the presentation of the UML models and to the animations, allowed to compare the effectiveness of both approaches in what concerns to their relative understandability.

This chapter describes how the technique presented in chapter 3, CPN-Tools and the BRITNeY Animation tool are recommended to be applied to support the building of an animation prototype for the uPAIN demonstration case.

4.1 Animations for requirements' validation

The effort to use only elements from the problem domain (external actors and use cases) in the user requirements models (use case and stereotyped sequence diagrams) and to avoid any reference to elements belonging to the solution domain (objects and methods) is not enough to obtain requirements models that are capable of being fully understandable to common stakeholders. This difficulty is mainly observable in what concerns the comprehension of the dynamic properties of the system within its interaction with the environment. This means that, even with the referred efforts, those static

requirements models should not be used to directly base the validation of the elicited user requirements by the stakeholders. Instead, we use those static requirements models to derivate animation prototypes.

User-friendly visualizations of the system behaviour, automatically translated from formal systems' models specifications, accepting user interaction for validation purposes, have been generically called animations. Despite seeming a good idea, in a context where IT is offering more and more powerful multimedia capabilities, the use of animation in user requirements validation, as a means of improving the understandability of systems' models by stakeholders, has been considered by only a small number of researchers.

In [Gemino, 2003] an empirical study has been carried out to comparatively evaluate the effectiveness of animation and narration (voice recordings of diagram explanations complemented with PowerPoint slides) in the process of communication of domain information to stakeholders for validation purposes, which may be seen as a sign that animation is increasingly drawing the software engineers' attention as a potentially valuable instrument for user requirements validation. The results of that empirical study were inconclusive about the effectiveness of animation, as opposed to the success of narration, but in our opinion that was due to the fact that, instead of using a meaningful user interface, the animations were of a very rudimentary type, simply consisting in highlighting the graphical elements of the diagrams while narration is being executed.

Some papers have been published, reporting the use of animations to ease validation by stakeholders, as is the case in [Fenkam *et al.*, 2002], where a *CORBA* API has been used to directly interpret *VDM-SL* specifications of requirements to generate a graphical user interface. Scenario-based approaches have also been used in [Ozcan *et al.*, 1998] as a means of ensuring user orientation, and also in [Uchitel *et al.*, 2004], where *fluents* (boolean system states that model pairs of system actions) have been used to relate goals with scenarios and, simultaneously, support animation. In [Winter *et al.*, 2001], virtual reality is used to support animation techniques when modeling high consequence systems (systems where errors in development have consequences of high cost).

4.2 Tools Integration

In the uPAIN project, the implementation of the interactive animation prototype demanded the usage of several technologies. The integration of tools was mainly based on XML files. Fig. 4.1 shows the global architecture of the tool environment used to generate the animation prototype. It is composed of a model executor and an animation tool. The model executor includes a *CP-net editor* and a *CP-net simulator*, both from CPN-Tools. The animation tool used corresponds to the *BRITNeY Animation tool* [BRITNeY].

With *BRITNeY Animation tool* it is possible to use pre-defined plug-ins (or write our own plug-ins) for executing some animation behaviour in the model. The pre-defined plug-ins include *SceneBeans* [Pryce and Magee] (an animation framework), message sequence charts (for displaying the passing of messages) and plot graphs. The writing of our own plug-ins involves the coding of Java classes and the creation of an XML description of the plug-in. *BRITNeY Animation tool* will automatically generate the code needed for the simulator to know of and use those plug-ins.

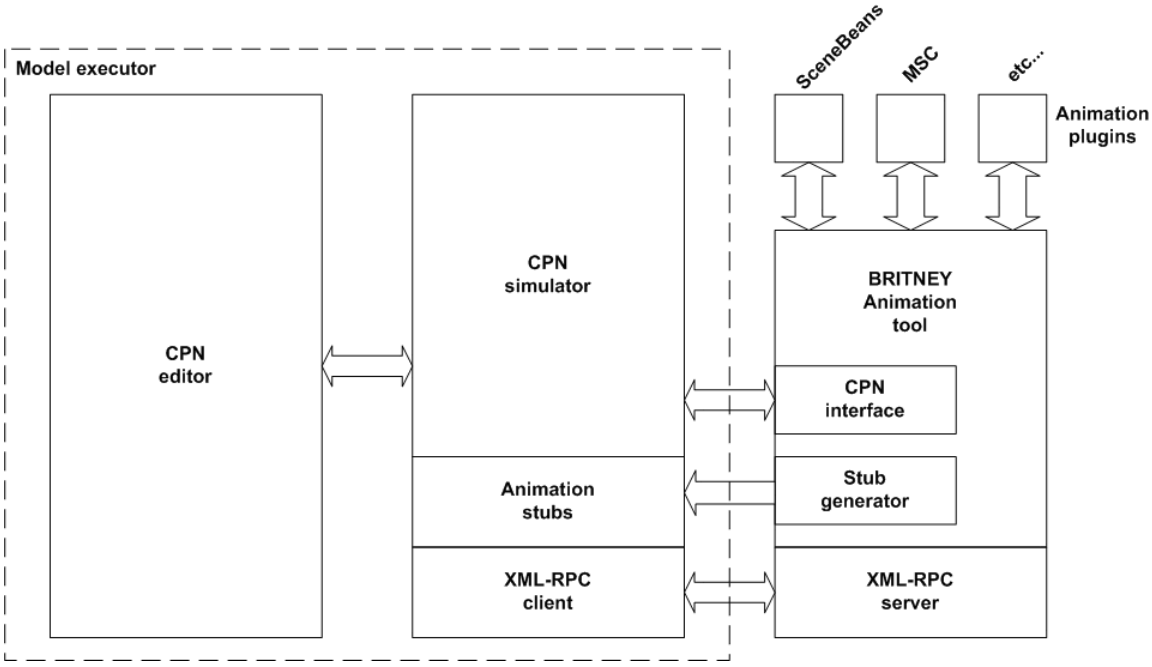

Fig. 4.1 – Global architecture for prototype animation.

It is possible to execute behaviours in the *BRITNeY Animation tool* while simulating models in CPN-Tools. Behaviours are executed through certain SML functions, which, in turn, call the corresponding Java methods. The names of the functions correspond to

those of the Java methods. When an SML function calls a Java method, it simply corresponds to the logic of an RMI call. The method name and arguments are passed over to the interface of the *BRITNeY Animation tool* and the return value of the executed method is passed back from the interface. If the method *M* in class *C* has the signature *int M (int x, string y)*, then it could be invoked as *C.M (42, "Hello World")*. However, this is just an example to explain how to use the Java methods in the CP-net model (see [BRITNeY] for complementary explanations). These behaviours, or methods, can be executed anywhere in the CP-net model where an expression is allowed. So, it can be on an arc expression, code segments on transitions (these are specific for CPN-Tools), and so on. This is a nice feature for debugging and for understanding the way the model affects the animation.

The *BRITNeY Animation tool* can also be executed as a standalone program, using e.g. Java WebStart to enable web browser integration. This feature is very useful to generate an autonomous animation prototype which allows stakeholders to "play with" without the interference and the presence of elements from the development team. This approach to validation was experimented with and proved to be very effective. This empowerment of the stakeholders promoted a deeper involvement of them in the analysis phase that not only assured better validation results, but also allowed the complementary elicitation of workflow requirements.

The interactive animation prototype for the uPAIN system is depicted in Fig. 4.2. The usage of *SceneBeans* allowed the animation of actors and message passing. *SceneBeans* provides a parser that translates XML documents into animation objects. A *SceneBeans* document is contained within a top-level <*animation*> element that contains five types of sub-elements: (1) a single <*draw*> element defines the scene graph to be rendered (e.g. the icons representing the doctor, the nurse, the patient); (2) <*define*> elements define named scene graph fragments that can be linked into the visible scene graph; (3) <*behaviour*> elements define behaviours that animate the scene graph (e.g. the animation of the drug injection from PCA icon to the patient icon); (4) <*event*> elements define the actions that the animation performs in response to internal events (e.g. the cleaning of the info text at the end of the drug injection animation); (5) <*command*> elements name a command that can be invoked upon the animation and define the actions taken in response to that command (e.g. the invocation of the behaviours responsible for the drug injection animation).

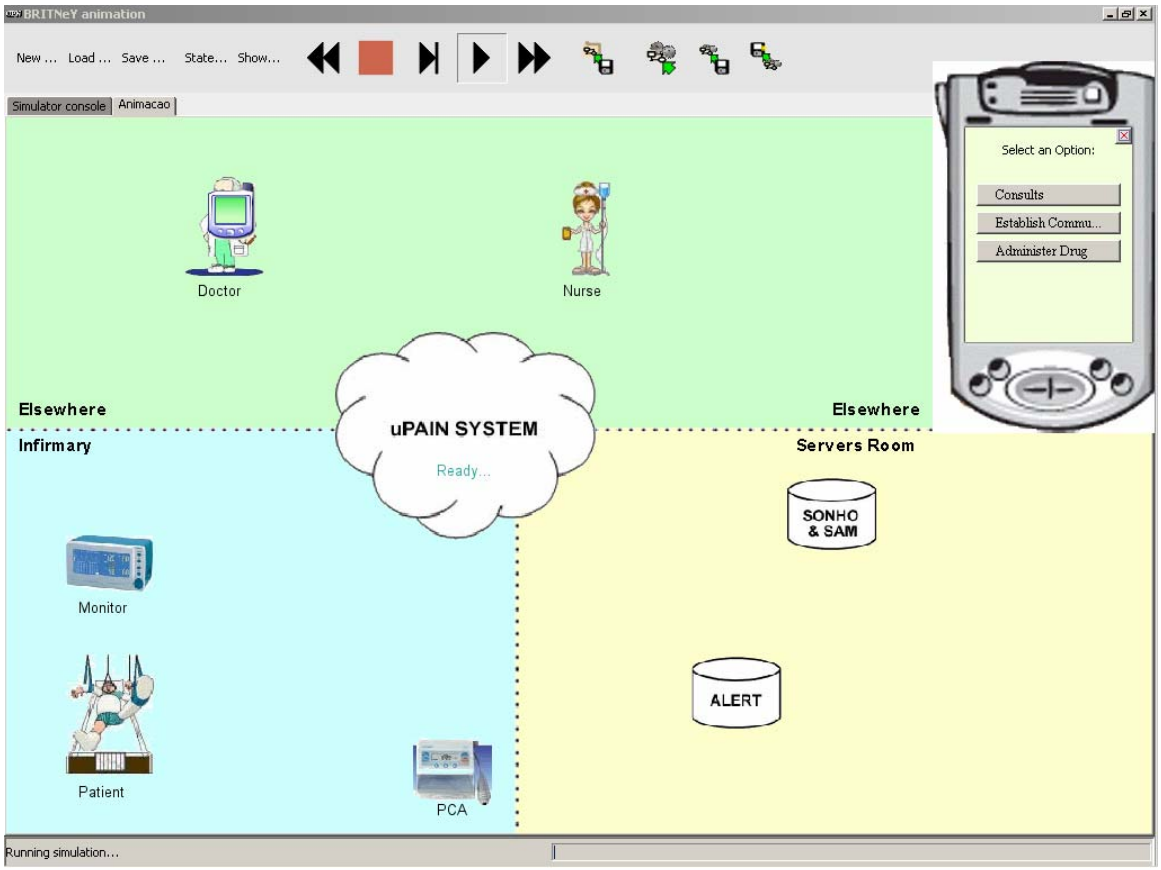

Fig. 4.2 – Interactive animation prototype for the uPAIN system.

Fig. 4.3 shows an example of a piece of code that was used in our <*draw*> element. This piece of code is responsible for the creation of the icons for the uPAIN system cloud (the first <*primitive*> element, inside the first *translate* type <*transform*> element), the patient (the second <*primitive*> element, inside the second *translate* type <*transform*> element, which, in turn, is inside an <*input*> element, because the patient icon works as a button) and the black ball (the third <*primitive*> element, inside the third *translate* type <*transform*> element) that represents the messages between actors. To animate the ball, we used the <*animate*> element, which means that parameters *x* and *y* will be animated by the behaviours *xf_patient_to_system* and *yf_patient_to_system*, respectively.

Fig. 4.4 shows the behaviours, the commands and the events that are responsible for moving the ball from the patient icon to the uPAIN system icon. When the simulator invokes the command *f_patient_t_system_cmd*, the <*reset*> and <*start*> elements execute the behaviours corresponding to the movement of the ball and the displaying of its textual info. When the execution of a behaviour ends, the animation will trigger the associated event, and this will start other behaviours, like *xball_out* (to hide the ball), *fadeout_info* (to

hide the textual info), or *hide_patientpda_icon* (to hide the patient PDA icon). At the end, the <*annouce*> element announces the event. This last action is crucial, because it allows the *CP-net simulator* to capture the event.

```
<draw>
...
<transform type="translate">
      <param name="translation" value="(${systemX},${systemY})"/>
     <transform type="scale">
     <param name="x" value="1"/>
     <param name="y" value="1"/>
       <primitive type="sprite">
           <param name="src" value="upainimages/system.gif"/>
       </primitive>
    </transform>
 </transform>
...
<input type="mouseClick" id="patient_button">
 <param name="releasedEvent" value="patient_button_pressed"/>
 <transform type="translate">
      <param name="translation" value="(${patientX},${patientY})"/>
       <primitive type="sprite">
           <param name="src" value="upainimages/patient.gif"/>
       </primitive>
 </transform>
 </input>
<transform type="translate" id="ball">
      <param name="translation" value="(-50,-200)"/>
      <animate param="x" behaviour="xf_patient_t_system"/>
      <animate param="y" behaviour="yf_patient_t_system"/>
      ...
      <primitive type="circle">
            <param name="radius" value="10"/>
      </primitive>
 </transform>
...
</draw>
```

Fig. 4.3 – Drawing in *SceneBeans*.

Communication between *SceneBeans* objects in the animation and the CP-net model can be done in two ways: (1) asynchronously, here the CP-net model simply invokes a command on a *SceneBeans* object and proceeds simulating, not caring for the moment when the animation behaviour that was executed terminates; (2) synchronously, here the CP-net model, again, invokes a command on a *SceneBeans* object, but, instead of just proceeding, the CP-net model waits for a particular event to arrive (e.g. the event “ball moved from patient to system”). This event would be broadcasted by the animation

command that was executed when it terminates to let the CP-net model know that this animation has completed.

```
<!-- behaviours -->
<co id="f_patient_t_system" event="msg_f_patient_t_system" state="stopped">
<behaviour algorithm="move" id="xf_patient_t_system">
    <param name="from" value="${patientX}+40"/>
    <param name="to" value="${systemX}+60"/>
    <param name="duration" value="${ballSpeed}"/>
</behaviour>
<behaviour algorithm="move" id="yf_patient_t_system">
    <param name="from" value="${patientY}+70"/>
    <param name="to" value="${systemY}+60"/>
    <param name="duration" value="${ballSpeed}"/>
</behaviour>
</co>
<!-- commands -->
<command name="f_patient_t_system_cmd">
  <reset behaviour="f_patient_t_system"/>
  <start behaviour="f_patient_t_system"/>
  <set object="info" param="text" value="Receiving request..."/>
  <reset behaviour="fadein_info"/>
  <start behaviour="fadein_info"/>
</command>
<!-- events -->
<event object="f_patient_t_system" event="msg_f_patient_t_system">
  <reset behaviour="xball_out"/>
  <start behaviour="xball_out"/>
  <reset behaviour="fadeout_info"/>
  <start behaviour="fadeout_info"/>
  <reset behaviour="hide_patientpda_icon"/>
  <start behaviour="hide_patientpda_icon"/>
  <announce event="order_from_patient_t_system" />
</event>
```

Fig. 4.4 – Defining behaviours, commands, and events in *SceneBeans*.

Synchronous interactions with *SceneBeans* objects must be carefully analyzed; otherwise, animations that should be executed in sequence will be executed concurrently. It is necessary to determine which animation behaviours are to be completed before any other can proceed (synchronous) and which ones can occur in any order (asynchronous). Invocations on *SceneBeans* objects are asynchronous in the sense that, per default, they do not broadcast any event; this has to be specified in the *SceneBeans* XML specification.

After creating all the behaviours, commands and events that allow the animation to announce events and receive commands from the *CP-net simulator*, the next step is to create Java classes. *SceneBeans* have the limitation of not allowing the user to input dynamic contents. In fact, *SceneBeans* only allows the creation of animations, based on static behaviours, defined in an XML file. The Java classes we created are responsible

for showing the graphical interfaces of the PDAs and for sending the corresponding user (of the animation prototype) inputs to the *CP-net simulator*. For instance, the Log Window that shows all the messages sent between actors demanded the creation of a Java class (*Messenger*) that receives the messages from the *CP-net simulator*. To add a new message to the list of messages of the Log Window, we simply invoke *Messenger.createAndShowGUI("message")*. After creating the Java classes, an XML description must be constructed so that the *BRITNeY Animation tool* recognizes them as plug-ins (see Fig. 4.5).

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 0.3"
"http://jpf.sourceforge.net/plugin_0_3.dtd">
<plugin id="UPAIN" version="0.1.0">
        <requires>
                <import plugin-id="AnimationTools"/>
        </requires>
        <runtime>
                <library id="upain_jar" path="UPAIN.jar" type="code">
                        <export prefix="*"/>
                </library>
        </runtime>
        <extension plugin-id="AnimationTools" point-id="Animation" id="ScenarioSelector">
                <parameter id="class" value="views.ScenarioSelector"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Ppda">
                <parameter id="class" value="views.Ppda"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Dpda">
                <parameter id="class" value="views.Dpda"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Npda">
                <parameter id="class" value="views.Npda"/>
        </extension>
        <extension plugin-id="AnimationTools" point-id="Animation" id="Messenger">
                <parameter id="class" value="views.Messenger"/>
        </extension>
                <extension plugin-id="AnimationTools" point-id="Animation" id="Chat">
                <parameter id="class" value="views.Chat"/>
        </extension>
</plugin>
```

Fig. 4.5 – Defining Java classes as plug-ins of *BRITNeY Animation tool*.

To access public methods of previously written Java classes (*Chat*, *ScenarioSelector*, *Messenger*, *Ppda*, *Dpda*, and *Npda*) and to invoke commands of the *SceneBeans* object (object anim) from the CPN-Tools, plug-ins must be declared and instantiated as objects in the index of CPN-Tools.

Java classes in defined animation plug-ins can be instantiated through SML (Standard Meta Language) functors that *BRITNeY Animation tool* generates. SML functors are "abstract" SML structures which can be instantiated. A Java object is instantiated by, e.g., *structure anim = SceneBeans(val name = "Name")*, which instantiates an object from the *SceneBeans* class. Methods on the instantiated *anim* are accessed as public methods defined in the *SceneBeans* class. Another example is the function *SPO ( )* in the transition

Select Patient Options of Fig. 3.11 that contains the following code to invoke methods to our Java objects:

```
Messenger.cleanText ( );
Ppda.createAndShowGUI ("mainmenu");
Ppda.getValueString ( );
```

SceneBeans objects provide also some methods to control the animation. For instance, the calling of function *move ( )* in the CP-net of Fig. 3.10 consists in an invocation of the method *invokeCommand* to the *SceneBeans* object *anim* (Fig. 4.6), which is responsible for invoking the previously defined commands in the XML file (Fig. 4.4). In this case, the invoked command corresponds to the movement of the black ball between the actors of the animation.

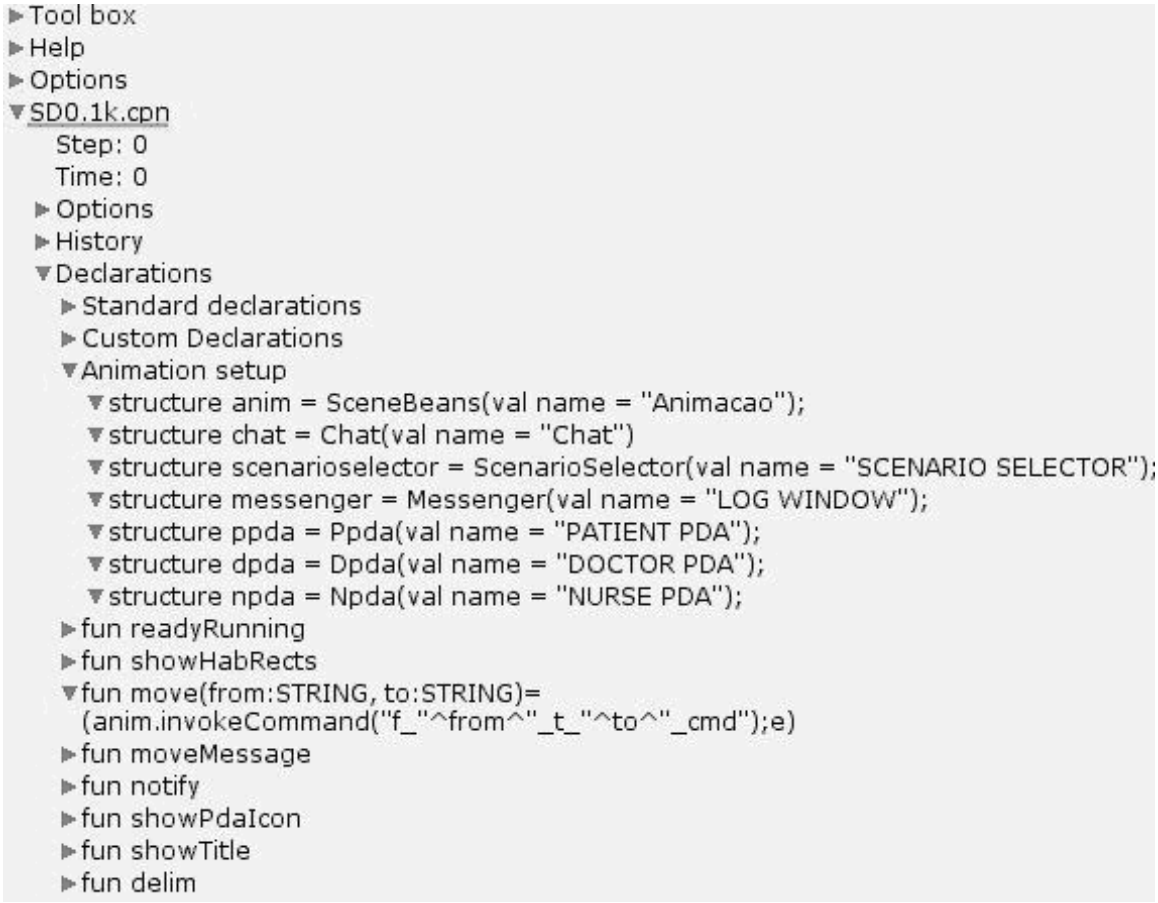

Fig. 4.6 – Declaring and instantiating objects in CPN-Tools.

Additionally, it is possible to capture events announced by the animation. In our animation prototype we included one CP-net subpage called *Events* (Fig. 4.8) that is composed by two distinct parts: one is responsible for the initial loading of the XML

animation description (places *Start* and *Running*, and transition *Init*); the other part includes the transition *Capture Event* (captures all the events announced by the *SceneBeans* animations and places them, in the form of a string list, in the place *Events*) and the place *Events*. This place *Events* belongs to a global fusion set of places that are connected to every transition where the capture of specific events is required (see, for instance, the nodes and arcs drawn with thinner lines in Figs. 3.11 and 3.12). All these fusion places are named *EventN* (where *N* is a digit that serves only as a distinguishing character, because CPN-Tools do not accept places with the same name, in the same page) and have no semantic meaning from the workflows' point of view. They are only needed for tool interoperability.

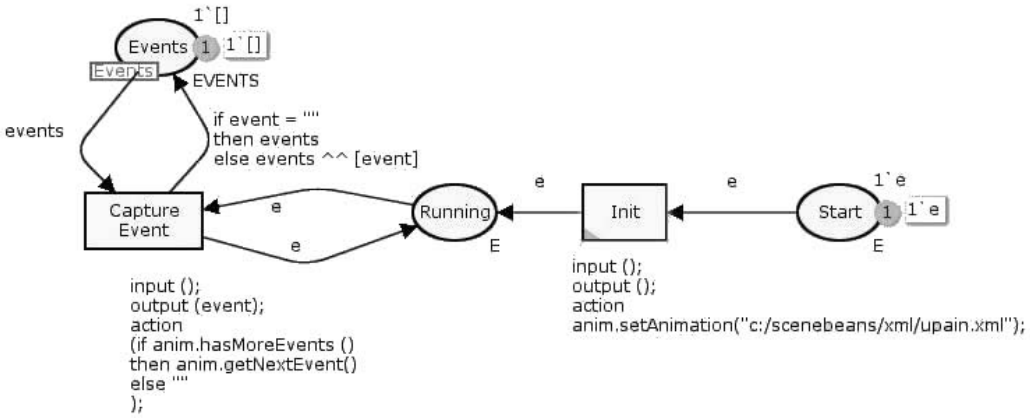

Fig. 4.7 – *Events* CP-net subpage.

4.3 Usability Issues

According to [ISO 9241-11], usability is considered "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". This means that, besides all the technical efforts described in the previous two sections of this chapter, the effectiveness of the implemented animation prototype to involve stakeholders in the interactive execution of the elicited sequence diagrams, complementary elicitation of workflow requirements and validation of the requirements model was also a result of a strong investment in using usability techniques in the construction of this software artefact, namely in what concerns its GUI (graphical user interface) and the comfort of exploitation of the animation prototype.

The adopted GUI makes use of eight icons on the display: three proactive actors (one patient, one medical doctor and one nurse); four reactive actors (one monitor, one PCA device and two databases in use at the hospital); and the uPAIN system represented by a cloud. The adopted GUI should be obvious and intuitive to the stakeholders and thus, with the exception of the cloud and the databases, we opted for "concrete" icons. When real world objects are represented in an icon ("concrete" icon), individuals are likely to find it more meaningful, are often familiar with the items depicted, and find it easy to make links between what is shown in the icon and the function it is supposed to represent [McDougall *et al.*, 2000]. To symbolize the uPAIN system (a concept which is difficult to materialize and to represent), we chose a cloud which constitutes an "abstract" icon. Forming strong systematic relations between icons and functions is very important, particularly when there are no pictorial alternatives for a given icon function [McDougall *et al.*, 2001]. To represent the uPAIN system we wanted an icon that emphasized its pervasive and wireless nature. Databases are also represented through an "abstract" icon that is a standard way to represent software technology databases. We also opted for uniform icons in terms of size because we wanted to avoid stakeholders focusing on some of the icons and not others due to size differences; we wanted them to have, at the first glance, the notion of the whole GUI. On the other hand, the real sized PDA is the biggest element and the only one that detaches from the GUI in terms of size, in order to improve the legibility of its contents.

Whenever one proactive actor is clicked with the mouse, a PDA icon appears above it and then, a real sized version of the PDA is also displayed, showing the predefined options, corresponding to possible requests (see Fig. 4.2). Through each proactive actor's PDA, the stakeholder just has to select the desired option and then the corresponding sequences are executed (each one of these is formally related with one of the UML stereotyped sequence diagrams).

Each time one of the proactive actors is clicked, a black ball (representing the actor's request) is sent from the actor towards the cloud. In Fig. 4.8, the stakeholder interacting with the animation prototype chose the "consult patient state" option by using the PDA of the medical doctor. The snapshot in Fig. 4.8 corresponds to the exact moment in which the monitor is sending to the uPAIN system some physiological indicators about the patient; this data exchange is graphically represented by the black ball trajectory in the display.

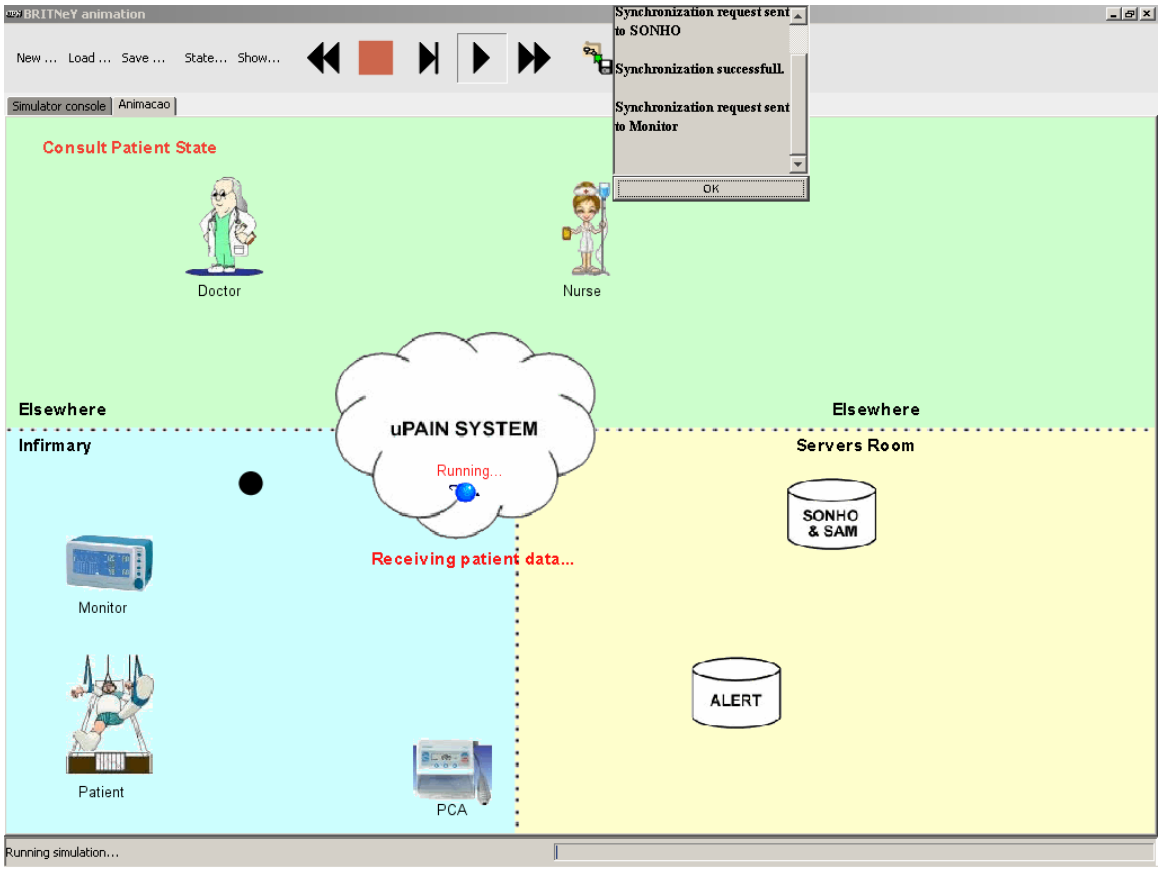

Fig. 4.8 – Message passing in the animation prototype for the uPAIN system.

This snapshot also shows a log window, where all the requests and interactions are registered. At the same time, underneath the cloud, a textual expression "receiving patient data" identifies the ongoing request/interaction. A caption, identifying the selected option is displayed during the whole action in the upper left corner of the display to prevent stakeholders from forgetting the task at hand and to provide them feedback, a golden rule of GUI design suggested in [Welie *et al.*, 1999]. It is crucial for stakeholders that the animation prototype lets them know at what point they are, at any given time in a clearly understandable way. Additionally, in Fig. 4.2 it is possible to observe a green colored "Ready…" message informing that the animation prototype is ready to accept one mouse click in one of the buttons of the displayed PDA. If, in any point of the simulation, the actor "uPAIN system" is in processing state, then a spinning globe appears inside the cloud and a red colored "Running…" message is presented (see Fig. 4.8).

To assure that the purpose of any graphical entity is clearly apparent and inferred (an important cognitive dimension in GUI design to deal with expressiveness [Pane,

2002]), a green dashed line contour was added around each proactive actor to make clear that only these are the proactive actors on which it is possible to click to produce some kind of interaction (Fig. 4.9).

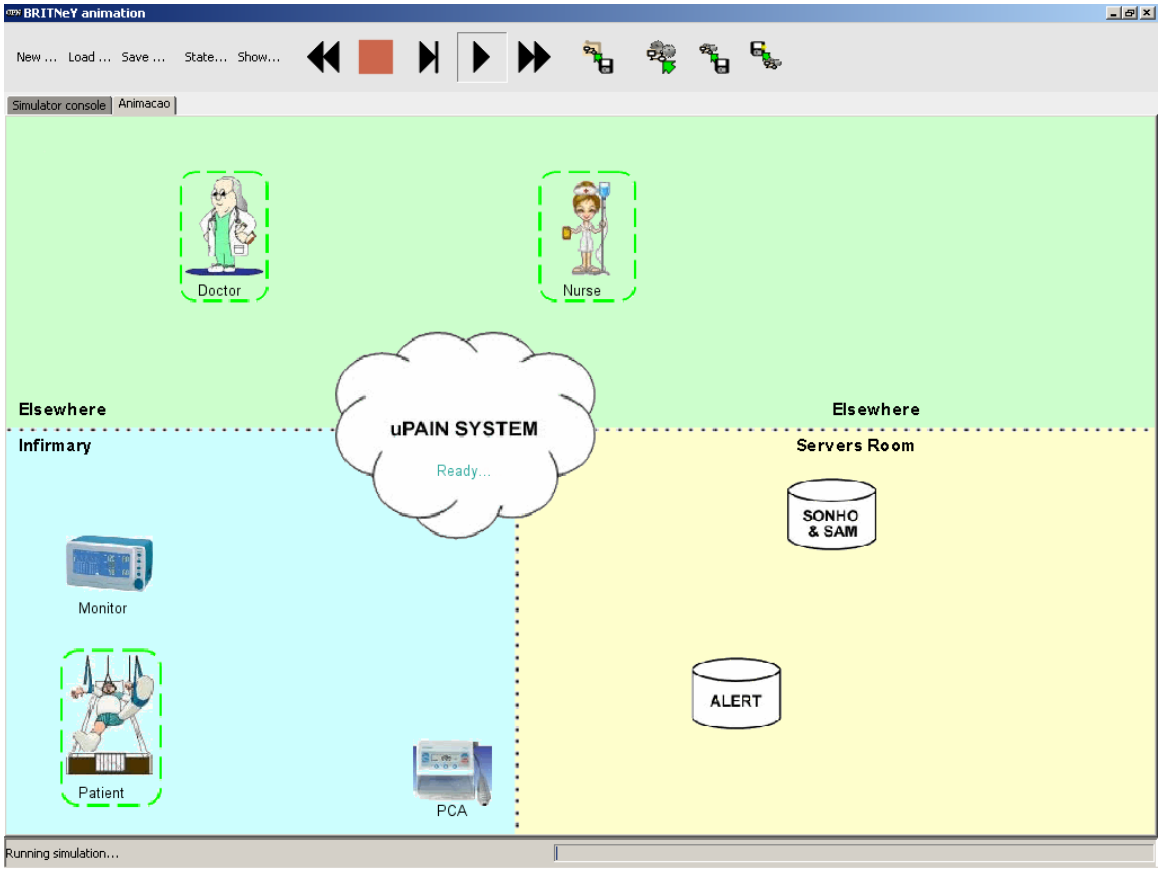

Fig. 4.9 – Dashed line contours in the animation prototype for the uPAIN system.

The green colored "Ready…" message also appears when these green dashed line contours are displayed. We also used a dashed line and different background colors to help delimit the three main areas of the GUI and grouping actors in a logical way, according to the areas in the hospital where they may be: the patient, the monitor and the PCA are always in the infirmary; the two databases are installed in the server's room; the medical doctor and the nurse can be elsewhere due to the nature of uPAIN system (ubiquitous); and the uPAIN system is "everywhere" in the hospital and so the cloud is placed in the middle of the three dashed areas. Below each actor, and to ensure that the actor is clearly identified immediately, the respective caption was added, since good labelling can guide stakeholders through the GUI with minimal search time. We also labelled the three dashed areas. This approach in GUI design contributes for a reduced cognitive load and immediate recognition in detriment of recalling in order to let stake-

holders make optimal use of their high-level cognitive abilities and save them to perform the essence of work; i.e., using the high level cognitive capacity for the more demanding work tasks such as workflow requirements validation, which is the real aim of the animation prototype.

The reduction of short-term memory load [Pane, 2002] was another intended goal, once in that part of memory only few information elements (typically, 5 to 8) can be stored simultaneously and the decay time is short (approximately 15 sec.). Thus, we avoided a dense area with many elements and presented only the necessary information.

4.4 Performance analysis

All that has been said so far in this dissertation about Petri nets has to do with the modeling of systems functionality. The term simulation has been used associated only to the execution of a CP-net, with the production and consumption of tokens, and the execution of code segments whenever transitions fire. Simulations of this kind allow the investigation of the logical correctness, functionality and dynamic properties of a system. However, during the simulation of a CP-net it is also possible to carry out performance analysis. The CP-net model was extended [Jensen, 1997 – vol. 2] with a global clock and the possibility for a token to carry a time stamp (in addition to the token color), which tells when the token is ready to be used by a transition. More recently, as a result of the work reported in [Wells, 2002], new functionalities have been added to CPN-Tools (already available in version 2.0.0) that extend considerably the capabilities of performance analysis of a model. In this section, we present briefly these new functionalities.

During a simulation, a CP-net model can contain and generate a significant amount of quantitative data that can be used to evaluate the system's performance. In order to evaluate the performance of a system, by means of the simulation of a CP-net model, but without introducing changes in its structure for that specific purpose, additional artefacts are needed. The most important ones are *monitors*.

Monitors are mechanisms that are used to observe, inspect, control or modify a simulation of a CP-net. They can inspect the *states* (markings) and *events* (occurring transitions) of a simulation, and take appropriate actions based on the observations.

With monitors, there is an explicit separation between monitoring the behavior of a net, and modeling the behavior of the system.

4.5 Discussion and conclusions

In this chapter, the application of the technique presented in chapter 3, CPN-Tools and the BRITNeY Animation tool were applied to support the building of an animation prototype for the uPAIN demonstration case.

The current development state of CPN-Tools makes the construction of CP-nets a comfortable, and even pleasant, task. Their powerful capabilities for the adjustment of all the visual details allow a perfect shaping of the nets, which greatly contributes for their optimized legibility. Their capability to simulate the functionality of CP-nets has proven to be complete and quite adequate to drive our animations. Besides, with the recent monitoring facilities, CPN-Tools and CP-nets have become a very strong “team” in every aspect of systems modeling and analysis. The weak link in the tool set that was used in the uPAIN demonstration is on the side of the animation tools. The BRITNeY Animation tool worked adequately as an interface between a graphical animation and the CP-nets being simulated, but there is still an important lack of plug-ins to ease the construction of animations, namely in what concerns to the degree of automation. An interface that allowed to draw the animation elements without the need to specify them manually in XML would be a valuable improvement. Another aspect to consider is the important processing power required. As an indication of processing requirements, our experience showed that computers with Intel processors prior to Pentium 4 or Pentium M, even with an amount of RAM of 512 MB, were practically incapable of running the animations.

The animation prototype was first demonstrated to the stakeholders with a strong involvement of the developers to explain the main approach to its usage as a software artefact to support the early execution of functional requirements. After that, the stakeholders have been given a standalone version of the animation prototype. This usage of the animation prototype has enabled the effective validation of requirements, since stakeholders frequently generated change requests to incorporate new scenarios and to adjust others already elicited, which has definitively contributed to the rapid evolution of the requirements model maturity, prior to design phase. We believe the usability con-

cerns we adopted in designing the whole animation prototype was determinant to the success of the uPAIN project.

Chapter 5

Conclusions

Static requirements models should not be used to directly base the validation of the elicited user requirements by the stakeholders, since the effort to use only elements from the problem domain in the user requirements models and to avoid any reference to elements belonging to the solution domain is not enough to obtain requirements models that are capable of being fully understandable by common stakeholders. The stakeholders' comprehension of the dynamic properties of the system within its interaction with the environment is better assured if animation prototypes, formally deduced from the elicited static requirements models, are used.

The behaviour of the animation prototypes can be specified by using CP-nets rigorously translated from use case and stereotyped sequence diagrams. An effective execution of UML models can be achieved by using CPN-Tools to operationally implement the interaction with the stakeholders within their efforts to validate the previously elicited workflow requirements models. Presently, the referred transformations are executed manually, which can be considered a major drawback of the proposed approach when the system to animate is of large dimension, presenting a great number of use cases and a large amount of behavioural scenarios to transform into CP-nets.

The generation of standalone versions of the interactive animation prototypes motivates stakeholders to get a deeper involvement in the analysis phase (without the interference of the development team). Usability features of the animation prototypes must also be carefully studied and experimented, before reaching the final version of the prototype in supporting the interactive execution of the elicited sequence diagrams, complementary elicitation of workflow requirements and validation of the requirements

models. CPN-Tools and *BRITNeY Animation tool* should evolve to support better the transparent generation of this kind of standalone versions and to allow a simpler start-up of an animation.

As future work, we intend to automatically generate CP-net skeletons from workflows requirements models (use case and stereotyped sequence diagrams). Additionally, we will study the possibility of using CP-nets, constructed for specifying the behaviour of the animation prototype, to base the behavioural specification of the elements that will compose the architecture of the system within the design phase. If data-flow languages (such as LabVIEW, as described in [Machado and Fernandes, 2002; Machado and Fernandes, 2005]) are used to develop the semantic layer responsible for integrating the whole ubiquitous system (embedded and mobile devices, database accesses and a service-oriented architectural platform), the asynchronous nature of CP-nets will smooth the transition from analysis to design phases in what regards behavioural models.

References

[Aalst *et al.*, 2003] van der Aalst, W.M.P., Hofstede, A.H.M. and Weske, M., *Business Process Management: A Survey*, In Aalst, W.M.P., Hofstede, A.H.M. and Weske, M. (Eds.), Lecture Notes in Computer Science, Vol. 2678, pp. 1–12, Springer-Verlag, 2003.

[Aalst, 2003] van der Aalst, W.M.P., *Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype"*, unpublished paper, available at http://is.tm.tue.nl/research/patterns/download/pi-hype.pdf, 2003.

[Aalst, 2004] van der Aalst, W.M.P., *Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management*, In Desel, J., Reisig, W. and Rozenberg, G. (Eds.), Lecture Notes in Computer Science, Vol. 3098, pp. 1–65, Springer-Verlag, 2004.

[Battiston *et al.*, 1988] Battiston, E., De Cindio, F., Mauri, G., *OBJSA Nets: A Class of High-level Petri Nets Having Objects as Domains*, In: G. Rozenberg (ed.): Advances in Petri Nets 1988, Lecture Notes in Computer Science, Vol. 340, pp. 20-43, Springer-Verlag, 1988.

[Best and Koutny, 2004] Best, E. and Koutny, M., *Process Algebra: A Petri-Net-Oriented Tutorial*, In Desel, J., Reisig, W. and Rozenberg, G. (Eds.), Lecture Notes in Computer Science, Vol. 3098, pp. 180–209, Springer-Verlag, 2004.

[Billington, 1988] Billington, J., Wheeler, G., Wilbur-Ham, M., *Protean: A High-LevelPetri Net Tool for the Specification and Verification of Communication Protocols*, IEEE Transactions on Software Engineering, Special Issue on Tools for Computer Communication Systems, Vol. 14, pp. 301-316, 1988.

[Billington, 1989] Billington, J., *Many-sorted High-level Nets*, Proceedings of the Third International Workshop on Petri Nets and Performance Models, Kyoto, pp. 166-179, 1989.

[BPMG] Business Process Management Group, http://www.bpmg.org/.

[BRITNeY] *BRITNeY Animation tool*, available at wiki.daimi.au.dk/tincpn

[Cassandras, 1993] Cassandras, Christos G., *Discrete Event Systems*, Richard D. Irwin, Inc., Homewood, IL, 1993.

[COSA] http://www.cosa-bpm.com/Business_Process_Management.html.

[David, 2005] David, R., *Discrete, Continuous, and Hybrid Petri Nets*, Springer, Jan 1, 2005.

[Dimitrovici *et al.*, 1991] Dimitrovici, C., Hummert, U., Petrucci, L., *Semantics, Composition and Net Properties of Algebraic High-Level Nets*, In: G. Rozberg (ed.), Advances in Petri Nets 1991, Lecture Notes in Computer Science, Vol. 524, pp. 93-117, Springer-Verlag, 1991.

[Ellis, 1979] Ellis, C.A., *Information Control Nets: A Mathematical Model of Office Information Flow*, In Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems, pp. 225–240, Boulder, Colorado, ACM Press, 1979.

[Fenkam *et al.*, 2002] Fenkam, P., Gall, H., Jazyeri, M., *Visual Requirements Validation: Case Study in a Corba supported Environment*, IEEE Joint International Conference on Requirements Engineering (RE'2002), 2002.

[FLOWer] http://www.pallas-athena.com/.

[Gemino, 2003] Gemino, A., *Empirical Comparisons of Animation and Narration in Requirements Validation*, Requirements Engineering, Vol. 9, pp. 153-168, Springer-Verlag, November, 2003.

[Gerogiannis *et al.*, 1998] Gerogiannis, V., Kameas, A., Pintelas, P., *Comparative Study and Categorization of High-Level Petri Nets*, The Journal of Systems and Software, Vol. 43, pp. 133-160, Elsevier Science Inc., 1998.

[Hoare, 1978] Hoare, C.A.R., *Communicating Sequential Processes*, Communications Of the ACM, Vol. 21, pp. 666-677, 1978.

[Holt, 1985] Holt, A. W., *Coordination Technology and Petri Nets*, In G. Rozenberg, editor,Advances in Petri Nets 1985, Lecture Notes in Computer Science, Vol. 222, pp. 278–296, Springer-Verlag, Berlin, 1985.

[IEEE, 1990] IEEE 610.12 1990: *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[IEEE, 2004] IEEE, *Guide to the Software Engineering Body of Knowledge* – 2004 Version, IEEE Computer Society, 2004.

[ISO 9241-11] ISO 9241-11:1998, *Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11: Guidance on Usability*, 1998.

[Jensen, 1997] Jensen, K., *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Volumes 1-3. Monographs in Theoretical Computer Science, Springer-Verlag, 1992 1997.

[Krämer and Schmidt, 1987] Krämer, B., Schmidt, H.W., *Types and Modules for Net Spacifications*, In: K. Voss, H.J. Genrich, G. Rozenberg (eds.): Concurrency and Nets, Advances in Petri Nets, Springer-Verlag, pp.269-286, 1987.

[Krüger *et al.*, 1999] Krüger, I., Grosu, R., Scholz, P., Broy, M., *From MSCs to Statecharts*, In F.J. Rammig (Ed.), Distrib-uted and Parallel Embedded Systems, pp. 61 72, Kluwer Academic Publishers, 1999.

[Lafon *et al.*, 2001] Lafon, M.B., Mackay, W.E., Andersen, P., Janecek, P., Jensen, M., Lassen, M., Lund, K., Mortensen, K., Munck, S., Ratzer, A., Ravn, K., Christensen, S., Jensen, K., *CPN/Tools: A Post WIMP Interface for Editing and Simulating Colored Petri Nets*, 22nd International Conference on Applications and Theory of Petri Nets (ICATPN 2001), Newcastle upon Tyne, UK, June, 2001.

[Levis, 1989] Levis, A.H., *Generation of Architectures for Distributed Intelligence Systems*, Massachusetts Institute of Technology, Report LIDS-P-1849, 1989.

[Liang, 2003] Liang, Y., *From Use Cases to Classes: a Way of Building Object Model with UML*, Information and Software Technology, no. 45, pp. 83–93, 2003.

[Machado and Fernandes, 2001] Machado, R.J., Fernandes, J.M., A Petri Net Meta-Model to Develop Software Components for Embedded Systems, Proceedings of the 2nd IEEE/FME International Conference on Application of Concurrency to System Design - ACSD 2001, Newcastle Upon Tyne, U.K., June, 2001, IEEE Computer Society Press, pp. 113-122, 2001.

[Machado and Fernandes, 2002] Machado, R.J., Fernandes, J.M., *Heterogeneous Information Systems Integration: Organizations and Methodologies*, In M. Oivo, S. Komi Sirviö (Eds.), 4th International Conference on Product Focused Software Process Improvement (PROFES'02), Rovaniemi, Finland, Lecture Notes in Computer Science Series, Vol. 2559, pp. 629-643, Springer-Verlag, December, 2002.

[Machado and Fernandes, 2005] Machado, R.J., Fernandes, J.M., *Integration of Embedded Software with Corporate Information Systems*, In A. Rettberg, M.C. Zanella, F.J. Rammig (Eds.), From Specification to Embedded Systems Application, IFIP Series, Vol. 184, pp. 169-178, Springer-Verlag, September, 2005.

[Machado *et al.*, 1998] Machado, R.J., Fernandes, J.M., Proença, A.J., *Hierarchical Mechanisms for High-level Modeling and Simulation of Digital Systems*, Proceedings of the 5th IEEE International Conference on Electronics, Circuits and Systems - ICECS'98, Lisbon, Portugal, September, 1998, Vol. III, pp. 229-232, 1998.

[Machado *et al.*, 2005a] Machado, R.J., Ramos, I., Fernandes, J.M., *Specification of Requirements Models*, In A. Aurum and C. Wohlim (Eds.), Engineering and Managing Software Requirements, pp. 47-68, Springer-Verlag, July, 2005.

[Machado *et al.*, 2005b] Machado, R.J., Fernandes, J.M., Monteiro, P., Rodrigues, H., *Transformation of UML Models for Service Oriented Software Architectures*, 12th IEEE Int. Conference on the Engineering of Computer Based Systems (ECBS 2005), Greenbelt, Maryland, U.S.A., pp. 173 182, IEEE Computer Society Press, April, 2005.

[McDougall *et al.*, 2000] McDougall, S.J.P., Curry, M.B., de Bruijn, O., *Exploring the Effects of Icon Characteristics on User Performance: The Role of Icon Concreteness, Complexity, and Distinctiveness*, Journal of Experimental Psychology: Applied, Vol. 6, no. 4, pp. 291-306, 2000.

[McDougall *et al.*, 2001] McDougall, S.J.P., Curry, M.B., de Bruijn, O., *The Effects of Visual Information on Users` Mental Models: An Evaluation of Pathfinder Analysis as a Measure of Icon Usability*, International Journal of Cognitive Ergonomics, Vol. 5, no. 1, pp. 59-84, 2001.

[Milner, 1980] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[MQSeries] *IBM WebSphere MQ Workflow*, International Business Machines Corporation, http://www-306.ibm.com/software/integration/wmqwf/.

[Murata, 1989] Murata, T., *Petri Nets: Properties, Analysis and Applications*, Proceedings of the IEEE, Vol. 77, no. 4, April, 1989.

[Ozcan *et al.*, 1998] Ozcan, M.B., Parry, P.W., Morrey, I.C., Siddiqi, J., *Requirements Validation Based on the Visualisation of Executable Formal Specifications*. International Conference on Computer Software and Applications, pp. 381-386, Austria, IEEE CS Press, 1998.

[Pane, 2002] Pane, J.F., *A Programming System for Children that is Designed for Usability*, PhD Thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, USA, May, 2002.

[Peterson, 1981] Peterson, J.L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ., 1981.

[Petri, 1962] Petri, C.A., *Kommunikation mit Automaten*, Bonn, Institute fur Instrumentelle Mathematik, Schriften des IIm no. 2, Also in English translation: *Communication with Automata*, Tech. Report RADC-TR-65-377, Vol. 1, suppl. 1, Applied Data Research, Princeton, NJ, 1966.

[Pryce and Magee] Pryce, N., Magee, J., *SceneBeans: A Component Based Animation Framework for Java*, http://www-dse.doc.ic.ac.uk/Software/SceneBeans/.

[Reisig, 1985a] Reisig, W., *Petri Nets: An Introduction*, EACTS 4, Brauer, W., Rozenberg, G., Salomaa, A. (Eds.), Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[Reisig, 1985b] Reisig, W., *Petri Nets with Individual Tokens*, Theoretical Computer Science, Vol. 41, pp. 185-213, North-Holland, 1985.

[Reisig, 1991] Reisig, W., *Petri Nets and Algebraic Specifications*, Theoretical Computer Science, Vol. 80, pp. 1-34, North-Holland, 1991.

[Staffware] *TIBCO® Staffware Process Suite*, TIBCO Software Inc., http://www.staffware.com/software/bpm/staffware_processsuite.jsp.

[Suzuki and Murata, 1983] Suzuki, I., Murata, T., *A Method for Stepwise Refinement and Abstraction of Petri Nets*, Journal of Computer and System Science, Vol. 27, no. 1, pp. 51-76, August, 1983.

[Symons, 1978] Symons, F.J.W., *Modelling and Analysis of Comunication Protocols Using Numerical Petri Nets*, PhD. Dissertation, Report 152, Department of Electrical Engineering Science, University of Essex, Telecomunication Systems Group, 1978.

[Uchitel *et al.*, 2004] Uchitel, S., Chatley, R., Kramer, J., Magee, J., *Fluent based Animation: Exploiting the Relation between Goals and Scenarios for Requirements Validation*, 12th IEEE Requirements Engineering International Conference (RE'04), 2004.

[Valette, 1979] Valette, R., *Analysis of Petri Nets by Stepwise Refinements*, Journal of Computer and System Sciences, Vol. 18, pp. 35-46, 1979.

[Valette, 1993] Valette, R., *Les Reseaux de Petri*, L.A.A.S./C.N.R.S., Toulouse, France, 1993.

[Vautherin, 1987] Vautherin, J., *Parallel Systems Specifications with Coloured Petri Nets and Algebraic Specifications*, In: . Rozberg (ed.), Advances in Petri Nets 1987, Lecture Notes in Computer Science, Vol. 266, pp. 293-308, Springer-Verlag, 1987.

[Welie *et al.*, 1999] Welie, M., van der Veer, G., Eliëns, A., *Breaking Down Usability*, Interact 99, Edinburgh, Scotland, 1999.

[Wells, 2002] Wells, L., *Performance Analysis using Colored Petri Nets*, PhD Dissertation, Department of Computer Science, University of Aarhus, Denmark, 2002.

[WFMC, 1999] *Workflow Management Coalition, Terminology & Glossary*, Document Number WFMC-TC-1011, Document Status - Issue 3.0, Feb 99, http://www.wfmc.org/standards/standards.htm.

[Whittle *et al.*, 2005] Whittle, J., Kwan, R., Saboo, J., *From Scenarios To Code: An Air Traffic Control Case Study*, Software and Systems Modeling, Vol. 4, no. 1, pp. 71-93, Springer-Verlag, February, 2005.

[Winter *et al.*, 2001] Winter, V., Desovski, D., Cukic, B., *Virtual Environment Modeling for Requirements Validation of High Consequence Systems*, Proceedings of the IEEE International Conference on Requirements Engineering, pp. 23-30, 2001.

[Zisman, 1977] Zisman, M.D., *Representation, Specification and Automation of Office Procedures*, PhD dissertation, University of Pennsylvania, Warton School of Business, 1977.

[Zowghi and Coulin, 2005] Zowghi, D. and Coulin, C., *Requirements Elicitation: A Survey of Techniques, Approaches, and Tools*, In A. Aurum and C. Wohlim (Eds.), Engineering and Managing Software Requirements, pp. 19–46, Springer-Verlag, July, 2005.