Formal Methods in Robotics:

a Probabilistic Constraint Nets Approach

by

Matteo Santoro

A dissertation submitted in partial satisfaction

of the requirements for the degree of

Doctor of Philosophy

in

Computation and Information Sciences

at the

Mathematics Department

of the

University of Naples *Federico II*

Abstract

This thesis seeks to provide some concrete answers to the growing need for a common, comprehensive framework for autonomous robotics. In fact, in the last few years, the focus of the robotic research is increasingly on building robots that interact autonomously with people, and even assist disabled people through social interaction. This *new robotics* poses new tough challenges for researchers. Long before autonomous (assistive) robots will show up all around us, we need sound ideas and quantitative methods to asses both their reliability and our safety in this new scenario. We might be able at least to guarantee that the behavior of a robot satisfies a set of global constraints – e.g. a safe and bounded response to unexpected events – or that the robot will eventually always accomplish its own task no matter whether the environment is static or not and whether fully observed or not.

The Probabilistic Constraint Nets (PCNs) Framework, proposed by St-Aubin et al. (2006) and St-Aubin (2005), seems to be a first concrete answer to the above mentioned problems. While the mathematical foundations have already been built, much work had yet to be done in order to make the framework acceptable in the robotic community. My work took up where St–Aubin's thesis stopped. I contributed to the PCNs framework in several ways. First, I discussed extensively the benefits and some of the limitations of using PCNs as a formal modelling language for robotic systems. Furthermore, I investigated the relationships between learning and PCNs. As a result I show here that most of the computational tools usually deployed to build robotic architectures by means of some learning device can be effectively expressed by means of PCNs. In order to narrow the gap between theory and practice, I introduce the software package called PCNJ as an effective development tool for robotic researchers. Finally, I explored the possibility of introducing formal methods also in the context of Computer Vision methods for robotics.

Table of Contents

Chapter 1

Introduction and Motivation

1.1 Thesis Statement

Because robots are on the verge of leaving their *industrial cages* and they are now poised to enter our homes and workplaces, robotic researchers are going to face new tough challenges. Above all, they must overcome a number of potential difficulties in designing and modelling very complex robotic systems, while preserving two crucial properties: reliability and also harmlessness.

While many efforts are being produced to attack directly specific sub–problems, I believe that better results can be obtained in the long–term by developing a comprehensive, theoretical framework for the design of autonomous robotic systems. An effective integration and fusion of all the contributions from many disciplines will produce a result that is much better that the simple "ad hoc aggregate" of all the parts.

This thesis aims at providing convincing arguments in favor of Probabilistic Constraint Nets as a viable candidate for the framework we are searching for. I show that most of the computational tools usually deployed to build robotic architectures can be effectively expressed by means of PCNs. I discuss the advantages we can obtain with this kind of approach and the usefulness of a cross–fertilization between PCNs and the other formalisms. In order to narrow the gap between theory and practice, I introduce the software package called PCNJ as an effective development tool for robotic researchers. Finally, a number of concrete robotic problems are presented and it is shown how we can overcome some of the difficulties, related

to them, by using Probabilistic Constraint Nets and PCNJ.

1.2 New Concerns and Challenges in Robotics

"*Can we trust robots?*"

This is definitely a tricky question, yet I believe it is among the most challenging ones that robotic researchers are asked to answer in the next few years.

First of all, for the sake of clarity, allow me to spend a few words to say what the question is not about. It is not about fear and scare. We must try to keep real Robotics and fiction as far apart as possible. During the past 30 years, loads of books and movies have been warning us about the danger of a robotic "rebellion" against human beings. Indeed, we must admit they certainly caused a deep feeling against a wider use of robots in our everyday life. Actually, if we remain in the realm of fiction then Asimov's Three Laws of Robotics (Asimov 1942) may be enough to prevent us from being attacked deliberately from robots.

Unfortunately, the solution is not so straightforward, so we'd better move on to the real world again. We need scientifically grounded answers, and the first step is to understand better which is actually the problem. If we looked at the *robot* market more carefully we would realize that robots are leaving their *industrial cages* and are now poised to enter our homes and workplaces. This consideration leads us back to the initial question because, of course, many concerns may arise naturally about safety implications of (semi)autonomous robots sharing the same environment with human beings beyond the factory domain. Given this scenario, I believe that the initial question is ill–posed because it doesn't focus enough on the real problem. Indeed, I suggest changing it slightly:

"*Can we trust people that sell us a robot?*"

Although at a first glimpse it might seem completely different from the original one, this new question goes to the core of the same problem and has the further merit of bringing the

issue back to a scientific level. Moreover, it is definitely more demanding then the previous one because it implies that the hypothetical *robot retailers* – and, most important, the Research Institutions[1] that are behind them – must be able to guarantee at least the following two conditions:

1. the robot we are going to buy – and that is going to share our living space – has a proven track record of reliability or, at least, it is certainly not harmful.

2. given that we need the robot to accomplish a specific task for us, there must exist a kind of "guarantee certificate" that the robot will always eventually reach the goal state, no matter how much noisy and unpredictable the environment is.

Although these requirements are exactly what any piece of electronic equipment is required by law to guarantee when we buy it, they still sound quite strange for a robot. The reason for that lies on the policy adopted by robot manufacturers so far. In fact, nowadays robots are present massively in almost all factories because they have been shown very helpful in industrial applications like assemblage and carriage of goods and loads. Most of the time industrial robots are huge, metallic arms ending with strange hooks, clamps, harpoons or spray guns; they are fenced for safety and people are not allowed to enter inside while they are switched on. It seems that the standard solution to the problem of robot–human interaction was the safest one: let's try to prevent the interaction completely. Indeed, despite the introduction of these and many other, more sophisticated safety mechanisms, robots have caused many victims over the years: people have been crushed, hit on the head, welded, etc. Last year, there were 77 robot-related accidents in Britain alone, according to the British Health and Safety Executive.

What happened in the field of industrial Robotics can give us a clear picture of how difficult the problem is. To keep people separate from robots didn't work in controlled envi-

[1] Both private (such as, for example, Sony or Pioneer) and public (such as Universities worldwide).

ronments such as factories; it is very unlikely it will work in far less controlled environments such as kitchens and living rooms. The picture is not more reassuring for us if we look at what is being done in order to make the interaction safer in those cases we cannot completely prevent it. The most popular approach is to program the robot to avoid any contact with moving objects (and thus with people). Despite the fact that the approach can sound quite simplistic, some good results have been obtained. Moreover, we must acknowledge that the problem is much harder that it sounds mainly because a robot that simply avoids anything on his path is quite useless in many applications.

The problem of regulating the behavior of robots is even worse if we consider that they are being increasingly built on autonomous–learning mechanisms. As robots are becoming more complex and – in some sense – "smarter", they are less predictable and tend to go wrong in unforeseeable ways.

To summarize things, the general feeling within the robotic research community is that we definitely need something more sophisticated than the above solutions. Actually, several promising events happened during this last year, and it is very likely that things are going to change[2]. Many research groups are trying to make robots safer and, furthermore, several of them are promoting an intense preliminary debate about concerns and practical problems for socially interactive robots. A worth mentioning example – and maybe a first step towards a new deal within the *technological innovation landscape* – is the EU–founded ETHICBOTS project that aims at coordinating a multidisciplinary group of researchers[3] with the common purpose of identifying and analyzing techno–ethical issues related to the integration of human beings and artificial (software/hardware) entities.

I deeply believe that effective solutions will derive primarily from a severe criticism for

[2]The Economist – Technology Quarterly published an interesting article (Jun,8th,2006) about new trends of Robotics and related safety problems. The electronic version can be downloaded from: `http://www.economist.com/science/displayStory.cfm?Story_ID=7001829`

[3]There are contributions from artificial intelligence, robotics, anthropology, moral philosophy, philosophy of science, psychology, and cognitive science.

and a revision of current methodologies used to design robotic systems rather than from any technical, partial advance in preventing human–robot harmful interactions. Throughout this thesis I'm going to promote a wide discussion about new *architectural* guidelines for both robotic systems modelling and behavior specification/verification.

1.3 Design and Modelling of Hybrid Systems

In the previous section, I described the first motivation behind the present body of work. A second one, perhaps even more interesting and urgent, is related to the problem of hybrid systems design and modelling. Hybrid systems consist of interacting discrete and continuous components (Tomlin and Greenstreet 2002, Maler and Pnueli 2003). Practical examples of hybrid systems include, among others: elevator systems, electric power distributions, automated factories, air traffic control systems, autonomous space craft controls.

Robots, of course, are further examples of hybrid systems. Indeed, they are among the most complex ones and we still lack a coherent methodology to design them. In order to make it clear what I'm talking about, let me quote from this insightful definition of Robotics (Hallam and Bruyninckx 2006).

> [...] Robotics is to a large extent a science of integration, constructing (models of) robotic systems using concepts, algorithms and components borrowed from various more fundamental sciences, such as physics and mathematics, control theory, artificial intelligence, mechanism design, sensor and actuator technology. The function and properties of a robotic system depend on the components from which it is made – the specific sensors, actuators, algorithms, mechanism – but, beyond that, they depend on the way those components are integrated. [...]

This definition draws an interesting picture of robotics as a melting pot of different disciplines that contribute to the development of these autonomous systems. Actually, I think

that the definition can be considered a road map to successful robotic system design because it clearly states the crucial importance of the overall architecture of the system beyond its constituent parts.

The focus of robotic research, then, should be (1) on finding new, effective architectural strategies and (2) on defining adequate specification languages that allow the system – and its properties as well – to be represented as a unified schema. In other words, a big amount of efforts might be devoted to studying formal models for hybrid systems, and the ultimate goal is to define structured formal languages for the specification of systems and their requirements and to develop methods for the verification of system behaviours.

In order to understand which kind of formal method we need, we must have a clear picture of the (hybrid) system under investigation. Thus, let's consider what a *robotic system* is, from a systemic point of view: basically it is a coupling of a *robotic agent*[4] to its *environment*. The robot itself comprises two distinct modules: a *body* which usually encompasses the various sensors and actuators, and a *controller*, which is usually a piece of software that controls the behavior of the agent.

With its sensors, the agent's body senses the environment, and reports to the controller what he perceived. The controller, given the updated piece of information about the state of the environment, sends appropriate control signals to the actuators of the body to perform the required actions which change the environment.

Figure 1.1 is a pictorial representation of a robotic systems. It shows how the coupled agent and environment act on – and react to – each other in a closed-loop system evolving over time.

Many critical issues arise when we try to model such a system as a whole:

- all the circuits and most of the hardware (of the body) are analog;

[4]Throughout this thesis I use the terms *robotic agents*, *robot* or simply *agent* as synonymous.

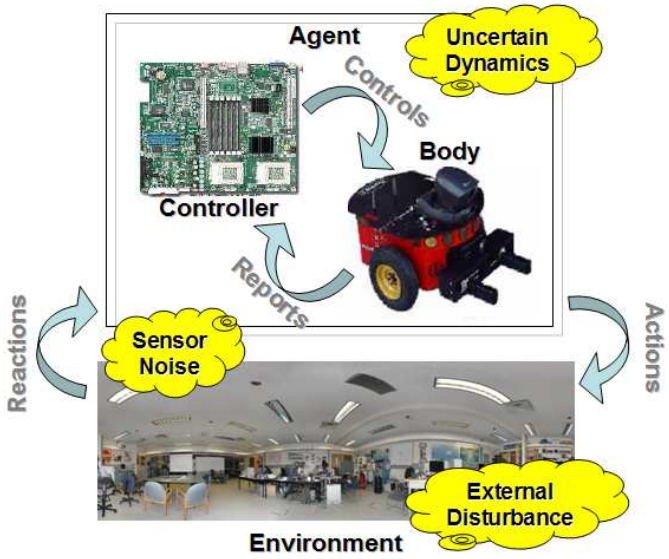


Figure 1.1: The structure of a constraint-based agent system

- controllers and software components that "govern" the behavior of the agent are (mostly) digital;
- the interaction between robot and environment is governed by a very complex dynamics that, due to the limitations in modelling of such systems, exhibits uncertainty and very often behaves probabilistically;
- various other types of uncertainty affect the system: for example, those originating from external disturbances, sensor noise and uncertainty in the correct execution of actions by the actuators;
- the closed-loop system of figure 1.1 comprises real objects that evolve in real time: we must be able to analyze the model in real time too.

This previous list refers to a lots of very difficult research problems and this somehow explains why only a few attempts have been made to develop formal methods for robotics. It is easier to attack specific subproblems, while hoping that the combination and coordination of all the results will come soon.

However, up to now there are already a few research groups concentrated on the topic of using formal methods for robotics systems. A book chapter has been devoted to a preliminary discussion about the topic (*Logics in Artificial Intelligence* 2004). Preliminary approaches to safety analysis were proposed also in (Seward et al. 1995); however they do not cover the verification and validation process. Leuschen et al. (1998) dealt with fault–tolerant robot architectures. Lankenau and Meyer (1999) proposed a fault–tree based method as a general verification approach for reactive systems. He emphasize the importance of employing formal methods for the design of robotic systems.

Although almost all the researchers do agree with the urge of formal methods in robotics – as exhaustively discussed above – this very short survey of the most relevant literature is a proof that this is still a pioneering research area. The first real, formal approach proposed so far is the Constraint Nets Framework and its stochastic generalization, on the path of which I've been working during the preparation of this thesis.

1.4 The Ins and Outs of my Research Work and Summary of Contributions

In the two previous sections I outlined a number of general, methodological problems that are emerging as central in robotics. The attempt to find suitable and concrete solutions to them has inspired my research work and this thesis from the very beginning. However, I acknowledge that such a long range goal may be considered quite pretentious and far beyond the scope of a single thesis. This is clear in my mind, and thus in this section I briefly reformu-

late the main purposes of the present dissertation by pointing out only the main contributions more precisely. They can be classified on three different levels:

1. (*methodological level*) to provide a (substantially) new point of view in the debate about methodologies for modelling autonomous robots;

2. (*theoretical level*) to contribute to the development of the Probabilistic Constraint Nets framework by discussing (1) the relationships between learning systems and PCNs; and (2) the the possibility of introducing formal methods also in the context of Computer Vision methods for robotics.

3. (*practical level*) to propose solutions to specific problems that are emerging within the research area of autonomous robotics; the most important contribution (at this level) is the development of PCNJ, an integrated development environment for people that want to design, build and "run" a PCN–based robotic architecture.

The work described in this thesis is well related to the research/study activity I've done as Ph.D. student. In fact, during Ph.D. program I have been studying a quite wide spectrum of research topics that goes from Computer Vision to Robotics through Machine Learning. The experience I accumulated in these areas provided me with the idea that most of the major advances in Robotics will be more related to *architectural features* than to specific subparts of the system.

Throughout this thesis I try to balance the description of different *tools*[5] – borrowed from different research fields – against the proposal of formal methods[6] and new, effective design approaches for robotics. I tried to link every practical solution to its methodological counterpart in the attempt to provide insightful elements for the general discussion about the modelling and critical systems.

[5] Both conceptual and practical.
[6] The ones based on the Probabilistic Constraint Nets framework.

1.5 Plan of the thesis

The thesis is organized as follows:

Chapter 2 : I present the basics of the Probabilistic Constraint Nets (PCNs) framework that has been originally introduced by St-Aubin (2005) to model any kind of stochastic, hybrid dynamical system. Even if the usefulness of the framework extends far beyond the scope of robotic research field, I believe that Robotics is the natural domain for exploiting PCNs and thus this chapter focus on some of the most interesting issues that can be useful in Robotics.

Chapter 3 : I summarize the notion of average–timed ∀–automata and discuss its links with behavioral specification and verification within the PCNs framework. Many mathematical details of the approach are omitted in order to guarantee a more general, conceptual understanding.

Chapter 4 : I look more carefully into the relationships between PCNs and several deterministic/probabilistic modeling frameworks commonly used in Robotics. I show that they can be considered as special cases of the PCNs framework, by providing – for each model – the PCN that computes exactly the same thing, i.e., the proposed PCN actually preserves the semantics of the computation.

Chapter 5 : I describe an integrated programming environment called PCNJ – that stands for *Probabilistic Constraint Nets in Java* – which supports probabilistic constraint net modelling, simulation, and animation for any kind of hybrid systems.

Chapter 6 : I discuss some concrete applications and problems that are relevant to the research on autonomous robotics. For each problem I propose a PCN–based solution, and furthermore I discuss interesting implications resulting from it. More specifically, I focus on problems arising in two broad areas of robotics: they are (1) behavior–based

motor coordination of mobile robots and (2) object recognition and localization for camera–equipped robots.

Chapter 2

Probabilistic Constraint Nets Framework

In this chapter I present the basics of the Probabilistic Constraint Nets (PCNs) framework that has been originally introduced by St-Aubin (2005) to model any kind of stochastic, hybrid dynamical system. Even if the usefulness of the framework extends far beyond the scope of robotic research field, I believe that Robotics is the natural domain for exploiting PCNs and thus this chapter and the following ones focus mainly on some of the most valuable contributions of PCNs to Robotics.

PCNs formalism is built on a topological, measure–based description of both time and domain structures. This abstraction is the main strength of the framework because it makes it possible: (1) to model time and domains as either discrete, continuous or hybrid structures, and (2) to describe uncertainty within the system appropriately (i.e. avoiding unwarranted over–simplifications of the model). This flexibility of the framework is a great asset as it allows the designer to describe complex systems under the umbrella of a single modelling language.

Since it is far beyond the scope of my thesis to discuss and demonstrate all the theorems and properties of the formalism, I refer the reader to the original work (St-Aubin 2005) for a more thoroughly description of PCN framework. Therefore I focus on those aspects that are more related to my own work and are essential for the overall comprehension.

This chapter is organized as follows: an informal discussion about the motivations that led

to the current formulation of PCNs framework is in section 2.2.1. The formal syntax of PCNs is then described in section 2.2.2 where a number of examples are provided in order to make it clear how it is possible to build a PCN–based model given a concrete hybrid system. The formal semantics along with some insightful comments are in section 2.4. As already pointed out, PCNs framework is heavily based on quite a number of rigorous mathematical concepts and theorems which the reader should be familiar with. However, in order to make it easier to understand the topics discussed in the chapter some of the most important mathematical concepts are shortly summarized in section 5.1.

2.1 Mathematical Concepts

Let's start with some mathematical concepts on which both syntax and formal semantics of PCNs are based. I reproduced or adapted in this section some of the definitions of (St-Aubin 2005) and (Zhang 1994). The main properties of the underlaying mathematical structures are summarized without giving any formal demonstration. The reader can find a more comprehensive introduction to the required mathematical concepts in (Gemignani 1967, Hennessy 1988, Manes and Arbib 1986, Warga 1972) (for what concerns topology and metric spaces) or in (Billingsley 1986, Breiman 1968, Williams 1991) (measure and probability theory). Chapter 3 of (Zhang 1994) is a useful compendium for modelling dynamics without uncertainty while the extension to uncertain dynamics are well summarized in chapter 2 of (St-Aubin 2005).

2.1.1 Time

The first pillar of the PCNs framework is the concept of *time* and its evolution. In general, without loss of generality, we can think of time $\mathcal{T}$ as a totally ordered set with a minimal element that we call the "initial start time". Furthermore, associated with $\mathcal{T}$, we need a

suitable metric to compute “the distance between any two time points” and a measure to talk about “the duration of an interval of time”.

Formally, we can define:

Definition 2.1 (Time Structure) *An* abstract time structure *is a triple $\langle \mathcal{T}, d, \mu \rangle$, provided the following conditions hold:*

1. *$\mathcal{T}$ is a linearly ordered set $\langle \mathcal{T}, \leq \rangle$, and $\mathbf{0}$ denotes the least element;*

2. *$\langle \mathcal{T}, d \rangle$ forms a metric space and d satisfies the equality*

$$d(t_0, t_2) = d(t_0, t_1) + d(t_1, t_2) \quad \forall t_0 \leq t_1 \leq t_2.$$

   Furthermore the two sets $\{t | d(\mathbf{0}, t) \leq \tau\}$ and $\{t | d(\mathbf{0}, t) \geq \tau\}$ must have a greatest and a least element respectively, for all $0 \leq \tau \leq \sup\{d(\mathbf{0}, t) | t \in \mathcal{T}\}$;

3. *$\langle \mathcal{T}, \sigma, \mu \rangle$ forms a measure space, where σ denotes the Borel set of the metric topology associated with $\langle \mathcal{T}, d \rangle$ and μ is the corresponding Borel measure. If we consider the subsets $[t_1, t_2) = \{t | t_1 \leq t < t_2\}$, then μ must satisfy the inequality $\mu([t_1, t_2)) \equiv \mu([\mathbf{0}, t_2)) - \mu([\mathbf{0}, t_1)) \leq d(t_1, t_2)$.*

Very often, if no ambiguity arises, it is possible to use simply $\mathcal{T}$ to refer to the time structure $\langle \mathcal{T}, d, \mu \rangle$. The natural choice is to define $\mu([t_1, t_2))$ in terms of $d(t_1, t_2)$, even if this is not necessarily the case.

Given the previous definition, the notion of infinite time, as well as those of continuous and discrete time can be stated formally:

- A time structure $\mathcal{T}$ is *infinite* iff $\mathcal{T}$ has no greatest element and $\mu(\mathcal{T}) = \infty$.

- A time structure $\mathcal{T}$ is *continuous* iff its metric space is connected.

- A time structure $\mathcal{T}$ is *discrete* iff its metric topology is discrete.

It can be easily shown that the set of natural numbers $\mathbb{N}$ along with the standard metric $d(t_1, t_2) = |t_2 - t_1|$ and the measure $\mu([\mathbf{0}, t)) = t$ is an example of infinite, discrete time structure. Both the same metric and measure can be defined on the set of non negative real numbers $\mathbb{R}^+$ in order to obtain a continuous time structure. Disconnected sets – like the union of intervals $I \subseteq \mathbb{R}^+$ – form time structures that are neither discrete nor continuous when they are equipped with the same metric and measure defined above.

The relationship between two different time structures is a further, worthwhile issue to discuss because it is related to the notion of *reference time mapping*. Let $\langle \mathcal{T}, d, \mu \rangle$ and $\langle \mathcal{T}_r, d_r, \mu_r \rangle$ be two time structures, we will say that $\mathcal{T}_r$ is the *reference time* of $\mathcal{T}$ – and that $\mathcal{T}$ is the *sample time* of $\mathcal{T}_r$ – is there exists a mapping $h : \mathcal{T} \to \mathcal{T}_r$ satisfying the following properties:

- the order among time points is preserved; i.e. $t < t'$ implies $h(t) <_r h(t')$,
- the least element is preserved; i.e. $h(\mathbf{0}) = \mathbf{0}_r$,
- the distance between two time points is preserved; i.e. $d(t_1, t_2) = d_r(h(t_1), h(t_2))$, and
- the measure on any finite time interval is preserved; i.e. $\mu([\mathbf{0}, t)) = \mu_r([\mathbf{0}_r, h(t)))$.

For example, $\mathbb{R}^+$ becomes the *reference time* of $\mathbb{N}$ if we define a mapping $h : \mathbb{N} \to \mathbb{R}^+$ so that $h(n) = n$. The notion of reference time is useful for the event–based systems, as it will be clearer soon.

2.1.2 Domains

Now that we are equipped with the notion of abstract time structure, we need to formalize the concept of abstract domain structure too, so that we can define uniformly both discrete and continuous domains.

Intuitively, we can distinguish between two types of domains: *simple domains* and *composite domains*. The former denote basic data types, such as reals, integers, Booleans, and characters; while the latter are related to arrays, vectors, strings, etcetera. The formalization of simple domain is quite straightforward: basically, it is a well–defined set A of elements – so that we can we decide if an element a either belongs to A or is undefined in A – and a metrics d_A to compute the distance between any two elements of A. The specification of d_A induces directly a metric topology τ and a partial order relation $\leq_A$ on A; we can thus define formally a simple domain as either a pair $\langle A \cup \{\perp_A\}, d_A \rangle$ or a triple $\langle \bar{A}, \leq_{\bar{A}}, \tau \rangle$, where $\perp_A$ means undefined in A and $\bar{A} = A \cup \{\perp_A\}$. A composite domain is defined recursively based on simple domains since it is the product of a family I of domains. The family I can be either finite or infinite, and either countable or uncountable. In general we state the following:

Definition 2.2 (Domain) *The triple* $\langle \bar{A}, \leq_{\bar{A}}, \tau \rangle$ *is a* domain *iff:*

- *it is a simple domain; or*
- *it is a composite domain, i.e. it is the product of a family of domains* $\{\langle A_i, \leq_{A_I}, \tau_i \rangle\}_{i \in I}$ *such that* $\langle A, \leq_A \rangle$ *is the product partial order of the family of partial orders* $\{\langle A_i, \leq_{A_i} \rangle\}_{i \in I}$ *and* $\langle A, \tau \rangle$ *is the product space of the family of topological spaces* $\{\langle A_i, \tau_i \rangle\}_{i \in I}$.

Given such a broad definition of domain, we might ask how it is possible to manage the diversity among different types of data. Intuitively, for any domain its partial order topology characterizes the information hierarchies of data and its derived metric topology characterizes the limit properties of data. Furthermore, as it will be clearer very soon, the PCNs framework relies on the concept of transductions that are mathematical models of general transformational processes; and thus we need a syntactical structure for specifying data types associated with such functions. The following two definitions are introduced for this purpose.

Definition 2.3 (Signature) *A* signature Σ *is a pair* $\langle S, F \rangle$ *where* S *is a set of sorts and* F *is a set of* function symbols, *provided that:*

- *F is equipped with a* mapping type*: $F \rightarrow S^* \times S$ where S^* denotes the set of all finite tuples of S;*
- *for any $f \in F$, type(f) is the type of f, i.e. type$(f) = \langle s^*, s \rangle$ means $f : s^* \rightarrow s$.*

A domain structure of a signature Σ is defined as follows.

Definition 2.4 (Σ–domain structure) *Let $\Sigma = \langle S, F \rangle$ be a signature. A* Σ–domain structure *is a pair $\langle \{A_s\}_{s \in S}, \{f^A\}_{f \in F} \rangle$ where for each $s \in S$, A_s is a domain of sort s, and for each $f \in F$ ($f : s^* - rightarrows$), $f^A : \times_I A_{s_i^*} \rightarrow A_s$ is a function denoted by f, which is continuous in the partial order topology.* ■

2.1.3 Traces and Events

Traces are functions from a time structure $\mathcal{T}$ to a domain A of values. They can be represented as a mapping $v : \mathcal{T} \rightarrow A$. A special type of trace is the so called *event trace* $e_{\mathcal{T}} : \mathcal{T} \rightarrow B$ whose domain B is a boolean set with only two distinct elements (e.g. 0 and 1).

In the PCNs framework the concept of trace is a crucial one because it allows us to describe the evolution of physical variables over time. Moreover, the notion of event trace provides a connection between continuous and discrete time structures. In fact, we can define an *event* as a transition either form 0 to 1 or from 0 to 1 of some event trace, and then we introduce the *event–based time* that is the set of all events in the trace. The time domain of an event trace is, of course, the *reference time* of the event–based time.

Unfortunately, simple traces and event traces are not suitable for dynamical systems that encompass uncertainty. For this purpose we must generalize the notion of trace to that of *stochastic trace* in order to be able to describe random changes of values over time. Formally, a *stochastic trace* is a mapping $v : \Omega \times \mathcal{T} \rightarrow A$ from a sample space Ω and a time domain $\mathcal{T}$ to the value domain A. It is easy to see that, for a given $\omega \in \Omega$, the function $v_\omega : \mathcal{T} \rightarrow A$

satisfy the above definition of trace[1].

As usual, in the presence of uncertainty we are more interested in the distribution of a system rather than in one specific configuration sampled form Ω. Thus, we prefer to look at the distribution of traces of a system rather than to pay attention to one given execution trace. In fact, the distribution of a stochastic trace provides complete information about the probability of the state of the system at every finite time point. However it is not possible to represent explicitly trace distribution values at infinite time points and so we must rely on the concept of limiting distribution of a stochastic trace because it assesses the behavior of the system in the long run. The following example will make this clearer.

Example 2.1 *Let's consider the system denoted by the equation:*

$$v(\omega, t) = 1 + B_t(\omega)e^{-t}, \tag{2.1}$$

where $B_t(\omega)$ is a Brownian motion process. It is straightforward to show that, for each t, v is normally distributed and $F_v = \mathcal{N}(1, te^{-2t})$. Intuitively, all the traces will exhibit a initial, transient stage of variability but then, as t increases, the spread of all the points is narrowed by the negative exponential term of the variance. In the long term all the possible traces will be undistinguishable and independent from the specific sample $\omega \in \Omega$. Formally, we can compute the limit distribution: $\lim_{t\to\infty} \mathcal{N}(1, te^{-2t}) = \mathcal{N}(1, 0)$*, which confirms the previous informal considerations. Figure 2.1 shows one specific execution trace of the system; the transient stage is pretty short and, after about* 500 *samples, the system converges to value* 1 *and doesn't fluctuate away from it any more, despite being influenced by a Brownian motion with increasing variance.*

Some mathematical difficulties may arise when we deal with limits in distributions of a stochastic trace v, i.e. v may not have an unique limit. This problem is discussed and solved

[1]We will use v to denote both the stochastic trace v and v_ω when no ambiguity caan arise

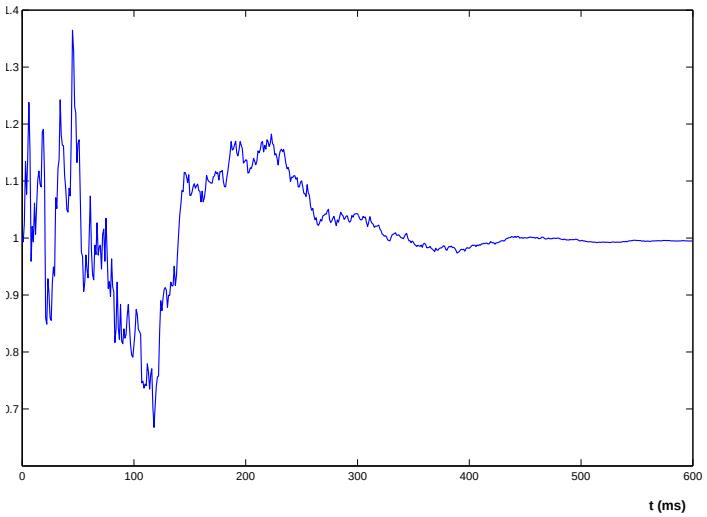


Figure 2.1: One specific execution trace of the system described by equation 2.1

in Chapter 2 of (St-Aubin 2005), to which the interested reader is referred.

The set of all the possible stochastic traces is named the *stochastic trace space* and will be a useful, synthesizing concept. The definition below formalize this concept as a composite domain so that we can use topological concepts to talk about limits.

Definition 2.5 (Stochastic Trace Space) *Given a time structure $\mathcal{T}$ and a domain $\langle A, \leq_A \ \tau \rangle$ the* stochastic trace space *is a triple $\langle A^{\Omega \times \mathcal{T}}, \leq_{A^{\Omega \times \mathcal{T}}}, \Gamma \rangle$ where $A^{\Omega \times \mathcal{T}}$ is the product set of all the function form $\Omega \times \mathcal{T}$ to A, $\leq_{A^{\Omega \times \mathcal{T}}}$ is the product partial order relation constructed form the partial order relation $\leq_A$, and Γ is the product topology constructed from the derived metric topology τ.*

In the following, a given trace space will be denoted $A^{\mathcal{T}}$ to simplify the notation when no ambiguity can arise.

Similarly to the deterministic case, it is possible to consider the special class of event–based stochastic traces that define sample time structures.

2.1.4 Transductions

A *transduction* represents a causal relationship between two stochastic trace spaces, i.e. it is a mapping from an input stochastic trace space to a corresponding output one. Roughly speaking, transductions dictate the evolution of a system by looking at the past and the current values of input traces. Several types of functional mappings actually meet these requirements, and thus it is possible to distinguish among a number of classes of transductions and even build a simple hierarchy.

A first huge difference exists between primitive transductions and event–based transductions. The former map stochastic traces to stochastic traces with the same time structure, while the latter can alter the time structure. A further distinction is between deterministic and probabilistic transductions depending on whether or not they encompass any kind of randomness.

If we look more carefully at the set of primitive transductions we can further refine the classification. In fact, generic primitive transductions comprise any functional composition of three basic elements: *i*) transliterations, which are memory–less combinational processes, *ii*) delays, and *iii*) generators, which allow for the modeling of uncertainty by introducing random variables in the model. Hierarchically built over the basic elements, compound transductions – either deterministic or probabilistic – can be defined by combining basic transductions of the same type with transliterations and delays.

Figure 2.2 schematizes the hierarchy of transduction types within the class of primitive transductions.

Finally, let's formalize the notions of basic primitive transductions.

Generators A transduction F is called a *generator* if it denotes a (potentially conditional) cumulative distribution function $F_{X|A}$ from which it can sample random variables at each time point. Formally, a generator is a function $\mathcal{G}^{A}_{\mathcal{T}}(v_0) : \Omega \times \mathcal{T} \times A \to A$ whose

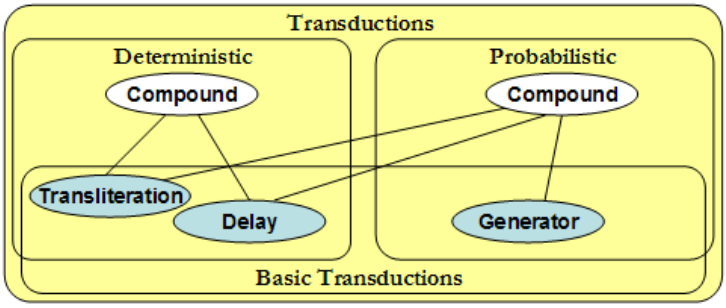


Figure 2.2: Types of primitive transductions and their reciprocal relationships

value is v_0 if $t = \mathbf{0}$ and $rand(F_{X|v(\omega,t)}(t), \omega)$ otherwise.

Transliterations A transduction F is called a *transliteration* if it is a pointwise extension of a function f. Intuitively, a transliteration is a transformational process without memory. Formally, if $f : \Omega \times A \to A'$ then its pointwise extension into a time structure $\mathcal{T}$ is a mapping $F : A^{\Omega\times\mathcal{T}} \to A'^{\Omega\times\mathcal{T}}$ so that $F(v)(\omega, t) = f(v(\omega, t))$

Delays A transduction δ is called a *delay* if it is a sequential process where the output value at any time is the input value at a previous time. Usually we distinguish between *unit* delays for discrete time and *transport* delays for continuous time. Let v_0 be a well–defined value in the domain A. If the time $\mathcal{T}$ is discrete, we use a unit delay $\delta^A_{\mathcal{T}}(\omega, v_0)(v)$ that is defined to be v_0 if $t = \mathbf{0}$ and $v(\omega, pre(t))$ otherwise. If the time $\mathcal{T}$ is continuous, a transport delay $\Delta^A_{\mathcal{T}}(\tau)(\omega, v_0)$ can be used and its value is v_0 is $m(t) < \tau$ and $v(\omega, t - \tau)$ otherwise.

Finally, let's consider the linkage between discrete and continuous components; it is modelled by event–driven transduction that can alter the time structure. More formally, an event–driven transduction is a transductions augmented with an extra input which is an event trace; event–driven transduction operate at every event and its output values holds form each event

to the next. A more rigorous mathematical presentation of the concept of event–driven transduction should go far beyond the scope of the present, explanatory section.

2.1.5 Dynamics Structures

We need a last mathematical entity before introducing syntax and semantics of PCNs; this is the *abstract structure of dynamics*.

Definition 2.6 (Σ–dynamics structure) *Let $\Sigma = \langle S, F \rangle$ be a signature. Given a Σ–domain structure A and a time structure $\mathcal{T}$, a* Σ–dynamics structure *$\mathcal{D}(\mathcal{T}, A)$ is a pair $\langle \mathcal{V}, \mathcal{F} \rangle$ such that*

- *$\mathcal{V} = \{A_s^{\Omega \times \mathcal{T}}\}_{s \in S} \cup \mathcal{E}^{\Omega \times \mathcal{T}}$ where $A_s^{\Omega \times \mathcal{T}}$ is a stochastic trace space of sort s and $\mathcal{E}^{\Omega \times \mathcal{T}}$ is the stochastic event space;*
- *$\mathcal{F} = \mathcal{F}_{\mathcal{T}} \cup \mathcal{F}_{\mathcal{T}}^{O}$ where $\mathcal{F}_{\mathcal{T}}$ is the set of basic transductions, including the set of transliterations $\{f_{\mathcal{T}}^{A}\}_{f \in F}$, the set of unit delays $\{\delta_{\mathcal{T}}^{A_s}(v_s)\}_{s \in S, v_s \in A_s}$, the set of transport delays $\{\Delta_{\mathcal{T}}^{A_s}(\tau)(v_s)\}_{s \in S, \tau > 0, v_s \in A_s}$, and the set of generators ; $\mathcal{F}_{\mathcal{T}}^{O}$ is the set of event–driven transductions derived from the set of basic transductions.*

2.2 Syntax of Probabilistic Constraint Nets

In this section the formal syntax of PCNs is introduced and a number of examples are proposed as a first step towards the understanding of the formalism. The section is divided into two parts; first, I show the rationale behind the proposed definition of PCN and then I state the syntax formally.

2.2.1 Informal Considerations Behind the Syntax of PCNs

Physical dynamical systems are sets of rules that describe the time dependence of physical variables. Mathematically, such rules are denoted by systems of equations, while the variables are represented as points in suitable geometrical spaces. The solutions of the equations tell us how the dynamical system evolves over time. Usually, the evolution rule of a system is given implicitly by a relation involving the future state as a function of the current state; these rules are referred to as differential equations and thus, because we are going to deal mostly with physical dynamical systems, PCNs syntax might be expressive enough to describe differential equations. On the opposite side, a second major class of systems that have been traditionally studied in the computer science community are discrete state machines. They are also known as digital systems and evolve by discrete changes between states. These discrete changes – or events – happen at certain time points and can be either synchronized or not. We want PCNs to able to describe this second class of systems too.

Finally, we ought to contemplate physical systems that consist of a mixture of interacting discrete and continuous components. These are known as *hybrid dynamical systems* (Tomlin and Greenstreet 2002, Maler and Pnueli 2003). Practical examples of hybrid systems include, among others: elevator systems, electric power distributions, automated factories, air traffic control systems, autonomous space craft controls and – most important from the point of view of the present thesis – robots. Hence an expressivity level suitable for hybrid dynamical systems is the ultimate aim of PCNs.

For this reason, Constraint Nets (CNs) have been originally proposed in Zhang (1994) as a formal method for designing and modelling hybrid dynamical systems. As a first important result of the proposed approach, Muyan-Ozcelik (2004) used CNs in order to show that the Constraint–Based Agent (CBA) framework with prioritized constraints is an effective methodology for designing and building Situated Agents (i.e. autonomous robots) in the real world. However, as the complexity of the task increases, it is not possible to ignore the unpre-

dictability and uncertainty of the robotic system – i.e. the robot coupled with its environment. Hence we might be able to model and analyze probabilistic systems.

CNs have a lot of nice properties that we'd like not to loose but unfortunately, they lack the ability of coping with uncertainties; in order to overcome this problem, Machworth and St.Aubin introduced PCNs as a non trivial extension of CNs so that, while keeping all the assets of CNs, they are also able to deal with unpredictable behaviors.

We are now equipped with an informal idea about PCNs and, most important, we know exactly what we should expect form them. Next subsection describes the formal syntax of PCNs.

2.2.2 Formal Syntax of PCNs

A Probabilistic Constraint Net is defined as follows:

Definition 2.7 (Probabilistic Constraint Nets) *A* Probabilistic Constraint Net *is a tuple* $PCN = \langle Lc, Td, Cn \rangle$*, where Lc is a finite set of locations, each associated with a sort; Td is a finite set of labels of transductions (either deterministic or probabilistic), each with an output port and a set of input ports, and each port is associated with a sort; Cn is a set of connections between locations and ports of the same sort, with the restrictions that* (1) *no location is isolated,* (2) *there is at most one output port connected to each location,* (3) *each port of a transduction connects to a unique location.* ■

Loosely speaking, we can think of locations as memory buffers in which it is possible to store the value of variables over time. Transductions are the functional elements of the system and can represent any kind of causal mapping – either deterministic or probabilistic – among locations. A transduction computes its output given the input over time and either operates according to a certain reference time or it is activated by external events. Connections define the relationship between locations and transductions. In order to be able to handle the uncer-

tainty in the systems, PCNS contain a special class of transductions that act as *generators*. Actually, they are random number generators that follow a give probability distribution.

Before discussing the properties of a PCN, it is useful to summarize the basics of PCN terminology:

- a location l is called an *output* location of a PCN iff l connects to the output port of a transduction in Td;
- a location l is called an *input* location iff it is not an output one. This follows from the observation that isolated locations are not allowed;
- $I(PCN)$ denotes the set of all input locations of a probabilistic constraint net PCN;
- similarly, $O(PCN)$ denotes the set of all output locations;
- a probabilistic constraint net is *open* if there exists at least one input location, otherwise it is said to be *closed*.

Many features of PCNs are easier to understand if we look at the representation of a PCN a graph. In fact, definition 2.7 induces a fairly simple graphical representation of a PCN as a bipartite graph whose vertices are either locations or transductions and whose edges are the connections. Edges can connect only vertex of one type to vertices of the other type. Locations are depicted by circles, transductions by boxes. In order to differentiate deterministic from probabilistic vertices, there is the convention of doubling the borders of the latter, that is to say generators are depicted by double boxes while random locations have double circles.

The following examples are the easiest way to fully understand both the definition 2.7 and the graphical representation. I use them to discuss a number of practical issues and point out some critical detail about PCN formalism more thoroughly.

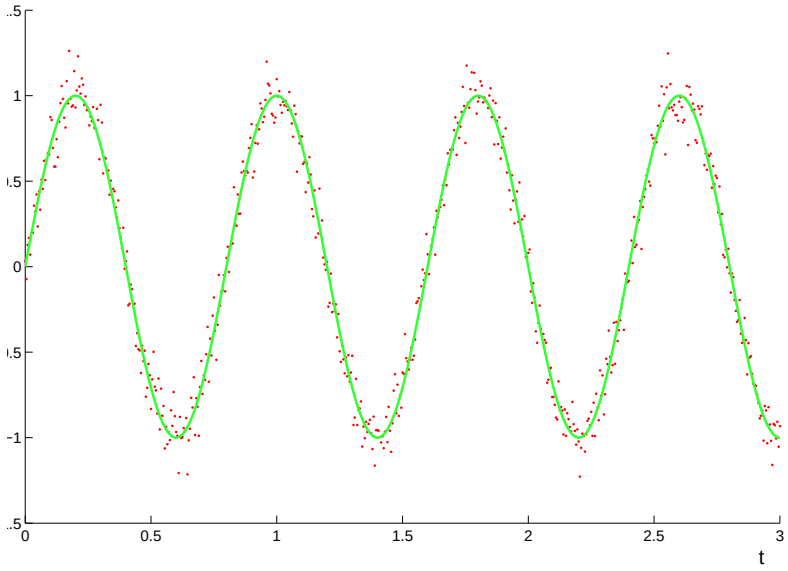


Figure 2.3: Red dots are sampled from the solution of equation 2.2 in the range $t = 0 \div 3s$; the sampling rate is 200 points per second. Green line represents the solution of $x(t) = \sin(t)$ in the same range and is plotted for comparison. The other parameters are: $\omega = 7,85 rad/s$; $\mu = 0$; $\sigma = 0,1$.

Example 2.2 (Simple equation with noise and explicit time dependence) *As a first example, let's consider the following equation:*

$$x(t) = \sin(\omega t) + N_{\mu,\sigma} \tag{2.2}$$

where t represent the time, ω is the angular frequency of the sinusoidal function and $N_{\mu,\sigma}$ is a random variable drawn – according to a gaussian probability distribution $G_{\mu,\sigma}$ – independently at each time instant t.

In figure 2.3 we plotted both the solution of $x(t) = \sin(\omega t)$ as a function of t (green line) in the range $t = 0 \div 3s$, and a sample solution of equation 2.2, drawn by means of a standard implementation of the random number generator $G_{\mu,\sigma}$.

This example is interesting because it allows me to discuss two important issues: first,

$x(t)$ in eq. 2.2 depends explicitly on time t; this is not the stantard case when we deal with dynamical systems. More usually the time is note explicitly represented in the syntax; i.e. we did not formally introduced the computation pipeline *yet. A PCN is a representation of a functional relationship between variables and the concept of traces of execution will come up in the next section in which the semantics is presented. From this point of view, the function $sin(t)$ is simply a function and not a well–defined* transliteration*.*

In order to overcome these problems, we must introduce a new variable T in the equation. The domain of T is actually our time structure $\mathcal{T}$. Let's use – in this case – a discrete time structure built on $\mathbb{N}$ with a fixed time step Δt between each time events. Thus, equation 2.2 becomes the following.

$$\begin{cases} x(T) = \sin(\omega T) + N_{\mu,\sigma} \\ T(n) = T(n-1) + \Delta t \end{cases} \tag{2.3}$$

The transduction sin is now a well defined transliteration that maps the input trace space defined by T into output the trace space defined by $x(T)$.

A second issue to discuss, is how we actually build the PCN model by starting from a given equation. In concrete, let's now build the PCN model (see fig. 2.4) of equation . We need at least the following set of variables, i.e. locations: $\{x, T, \Delta T, \mu, \sigma, N_{\mu,\sigma}, k, h, z\}$ where k, h, and z are temporary variables that won't appear in the interface of the PCN[2]*. The variable k stores the product $\omega * t$, h stores the value of $sin(k)$, and z stores the increment of the time variable and is therefore the input of a unit time delay; that is important in order to avoid algebraic loops in the NET*[3]*. The meaning of the other variables is obvious. All the variables are deterministic except for the output of gaussian generator: $N_{\mu,\sigma}$, that will be depicted with double circle.*

[2]Throughout this thesis I will omit to assign an explicit label to these variables since they are meaningless from outside the PCN.

[3]See Zhang (1994) and St-Aubin (2005) for more details about algebraic loops in PCNs.

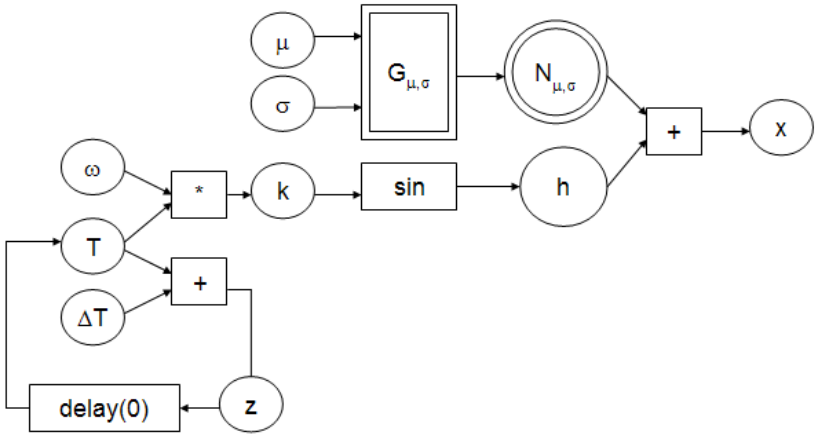


Figure 2.4: PCN representing equation 2.2

The transductions labels are: $\{+, *, delay(1), sin, G_{\mu,\sigma}\}$. *As you can see in figure 2.4, the label + is used twice in the PCN. Actually, the two transductions are distinct and they must be kept separated in order to guarantee the semantical correctness of net (as we'll se later). Whenever some confusion or even a mistake can arise, it is preferable to use two distinct labels: for example* $+_1$ *and* $+_2$.

The transduction labelled $G_{\mu,\sigma}$ *is doubly squared because it is a random number generator and, thus, it introduces a non determinism in the net.*

Before going any further, it is worth spending a few words about random locations: basically they are the output location of some generator. However, if we add one generator in our net, then its output is a random variable whose value varies according to the specified probability distribution. All the (deterministic) transduction with this (random) location as input will have an output that, in principle, follows itself a modified version of the same original probability distribution. Therefore it happens that in the presence of at least one generator all the locations that descend from it should be considered random location; furthermore if

we add a feedback then it can happen that all locations are random locations. The idea, thus, is to use double border only for locations that are output of a generator as it provides a visual and intuitive way of assessing where uncertainty initially enters in the system.

Example 2.3 (Simple Pendulum) *In this second example I show how it is possible to apply the definition 2.7 in order to build a PCNs–based model of a well–known physical system: the simple pendulum.*

A simple pendulum consists of an oscillating point mass attached to an inextensible weightless string. When displaced to an initial angle and released, the pendulum will swing back and forth with periodic motion. The equation of this physical system can be obtained by applying Newton's second law:

$$mL^2 \frac{d^2\theta}{dt^2} = -mgL \sin\theta, \tag{2.4}$$

where m is the mass, L is the length of the string, g is the gravitational acceleration and θ is the displacement angle.

Equation 2.4 can be reformulated in terms of the so–called resonant frequency $\omega \equiv \sqrt{g/L}$ and becomes:

$$\ddot{\theta} + \omega^2 \sin\theta = 0. \tag{2.5}$$

If the amplitude of displacement is small enough so that the small angle approximation *holds, i.e.* $\sin\theta \approx \theta$*, then the equation of motion reduces to the equation of simple harmonic motion:*

$$\ddot{\theta} + \omega_0^2 \theta = 0.$$

The simple harmonic solution is $\theta(t) = \theta_0 \cos(\omega t + \varphi)$.

However, if the angular displacement of the pendulum is large enough then the small angle approximation no longer holds and the equation of motion remains the 2.4. This differential equation does not have a closed–form analytical solution, and we have to rely on approximations, i.e. we must try to solve it numerically using a computer by means of some iterative method for solving differential equations. Here let's use the standard forward Euler method. This is a quite popular method for solving ordinary differential equations using the formula: $y_{n+1} = y_n + \Delta t f(y_n, t_n)$*, which advances a solution from* t_n *to* $t_{n+1} = t_n + \Delta t$*. In practice the method increments a solution through an interval* Δt *while using derivative information from only the beginning of the interval.*

As a first step if we want to use this method, let's translate the second order differential equation into the first order system of differential equations:

$$\begin{cases} \frac{d}{dt}\dot{\theta} &= -\omega^2 \sin\theta \\ \frac{d}{dt}\theta &= \dot{\theta} \end{cases} \tag{2.6}$$

For the first equation, thus, we have:

$$\frac{d}{dt}\dot{\theta} \approx \frac{\dot{\theta}(t + \Delta t) - \dot{\theta}(t)}{\Delta t}$$

where the equality hold only in the limit $\Delta t \to 0$*. The same approximation holds for the second equation.*

We are allowed to rewrite the system 2.6 as:

$$\begin{cases} \dot{\theta}_{n+1} = \dot{\theta}_n + \Delta t(-\omega^2 \sin\theta) \\ \theta_{n+1} = \theta_n + \Delta t\dot{\theta}. \end{cases} \tag{2.7}$$

It is easy to see that the PCN depicted in figure 2.5 represents the system2.7.

Example 2.4 (State Transition Systems) *This third example shows that two nets can denote*

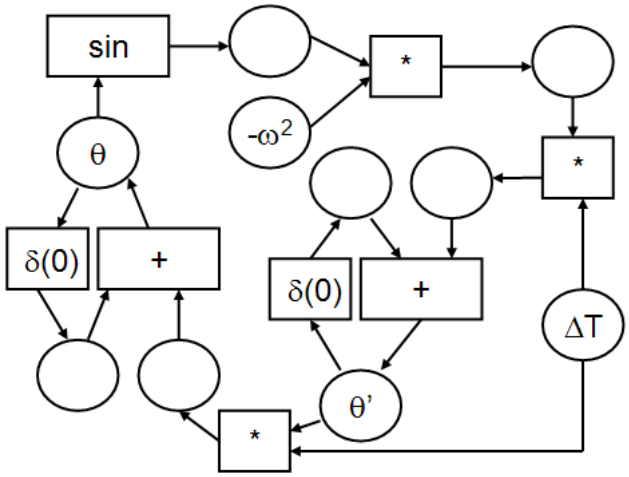


Figure 2.5: PCN representing system of equations 2.7

quite different dynamical systems even if they are extremely similar from a pictorial point of view.

Let us consider the graph in Figure 2.6 where f *is a generic transliteration and* δ *is a unit delay. If we suppose the time is discrete, then this net can be also written as the equations:* $s(n) = f(u(n-1), s(n-1)), s(0) = s_0$. *More simply, if we allow* s' *to denote the* next state *of* s*, we can write the equations as:* $s' = f(u, s), s(0) = s_0$.

If we consider continuous time and slightly modify the graph (see fig. 2.7) by letting the transliteration f *be the standard Riemann integral then we obtain the constraint net of an ordinary differential equation:* $\dot{s} = f(u, s), s(0) = s_0$.

2.3 Subnets, modules and hierarchical modelling

Complex physical systems may be composed of a set of subsystems which – by interacting together in a hierarchical fashion – produce the behavior of the global system. Thus, it is

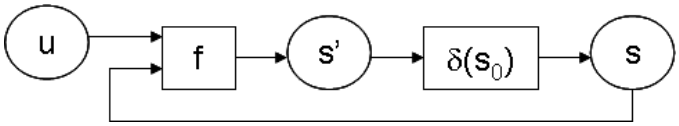


Figure 2.6: The constraint net representing a state transition system.

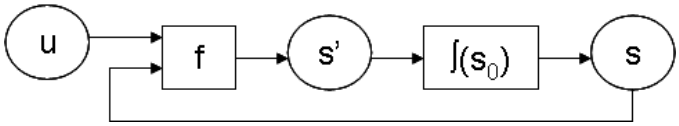


Figure 2.7: The constraint net representing a differential equation.

worth to define the two concepts of *subnet* of a PCN and of *module*; then we discuss how it is possible to compose different modules preserving all the properties of the PCN.

Definition 2.8 (Subnet) *A probabilistic constraint net* $PCN_1 = \langle Lc_1, Td_1, Cn_1 \rangle$ *is a subnet of* $PCN_2 = \langle Lc_2, Td_2, Cn_2 \rangle$*, written* $PCN_1 \subseteq PCN_2$ *iff* $Lc_1 \subseteq Lc_2$, $Td_1 \subseteq Td_2$, $Cn_1 \subseteq Cn_2$ *and* $I(PCN_1) \subseteq I(PCN_2)$. ■

Definition 2.9 (Module) *A module is a triple* $\langle PCN, I, O \rangle$*, where PCN is a probabilistic constraint net,* $I \subseteq I(PCN)$ *and* $O \subseteq O(PCN)$ *are subsets of the input and output locations of PCN. We say that* $I \cup O$ *defines the interface if the module. When it is clear by the context, we will use the notation* $PCN(I, O)$ *to denote the module* $\langle PCN, I, O \rangle$. ■

Graphically, a module will be represented by a box with rounded corners. Moreover, all the locations in $I(PCN) - I$ and $O(PCN) - O$ are respectively *hidden* inputs and *hidden* outputs and are used to model non–determinism in the system.

It is possible to introduce three basic operations to obtain a new module from existing ones. These are:

Union The union operation is used to obtain a new module created by two modules side by

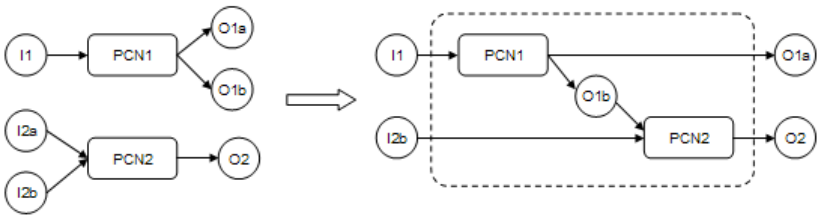
side. Formally, let $PCN_1 = \langle Lc_1, Td_1, Cn_1 \rangle$ and $PCN_2 = \langle Lc_2, Td_2, Cn_2 \rangle$ be two probabilistic constraint nets, with $Lc_1 \cap Lc_2 = \emptyset$ and $Td_1 \cap Td_2 = \emptyset$, then the union of $PCN_1(I_1, O_1)$ and $PCN_2(I_2, O_2)$, written $PCN_1(I_1, O_1) || PCN_2(I_2, O_2)$, is the new module $PCN = \langle Lc_1 \cup Lc_2, Td_1 \cup Td_2, Cn_1 \cup Cn_2 \rangle$, whose interface is defined by $I = I_1 \cup I_2$ and $O = O_1 \cup O_2$.

Coalescence The coalescence operator combines two locations in the interface of a module into one, with the restriction that at least one of these two locations is an input location. Formally, let $PCN = \langle Lc, Td, Cn \rangle$ be a probabilistic constraint net, $l \in I$ and $l' \in I \cup O$ be of the same sort, the coalescence of $PCN(I, O)$ for l and l', denoted $PCN(I, O)/l, l'$ is a new module $PCN'(I', O)$ with[4] $PCN' = \langle Lc[l'/l], Td, Cn[l'/l] \rangle$, $I' = I - \{l\}$.

Hiding The hiding operation deletes a location from the interface by turning it inot a hidden location. Formally, let $PCN = \langle Lc, Td, Cn \rangle$ be a probabilistic constraint net and $l \in I \cup O$, the hiding of $PCN(I, O)$ for l, denoted $PCN(I, O) \backslash l$, is a new module the module $PCN'(I', O')$ with $PCN' = PCN$, $I' = I - \{l\}$ and $O' = O - \{l\}$.

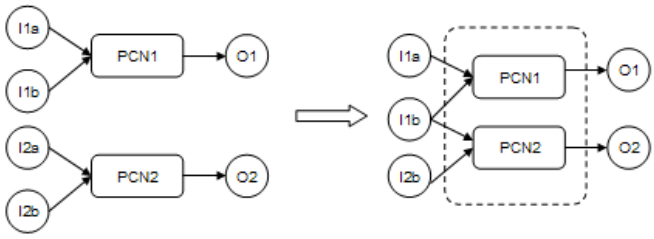
Furthermore, it is possible to define three combined operations:

Cascade The cascade connection connects two modules in series.

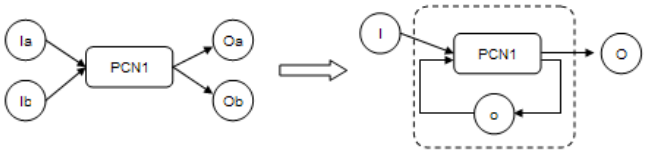


Parallel The parallel connection connects two modules in parallel.

[4] $X[v/x]$ denotes that x in X is replaced by v



Feedback The feedback connection connects an output of the module to an input of its own.



2.4 Semantics of Probabilistic Constraint Nets

In this section I define the formal semantics of probabilistic constraint nets, which is necessary to provide a meaningful way to interpret PCNs. It happens that the formal syntax of PCNs is quite similar to several other formal models – e.g. Petri Nets (Peterson 1981) and their generalization *Colored* Petri Nets (Jensen 1981) – which also have been proposed as formal language for dynamical systems. After this section it should be clear that, despite these models share many of the syntactical features, they have completely different semantics and no confusion may arise.

St–Aubin proposed to define the formal semantics of PCNS by using the fixpoint theory which is a common approach to describe the semantics of programming languages[5]. The general idea behind such an approach is quite simple: a program defines a function f and its semantics is defined to be the least solution of $x = f(x)$ that is to say the least fixpoint of f. Because any PCN is a set of equations with location serving as variables, the application of

[5]This choice is consistent with the approach adopted by Zhang and Mackworth (1995) for Constraint Nets.

this theory should seem quite straightforward: the semantics of a PCN can be the least fixpoint of the set of equations. Unfortunately, some of the variables in the equations are supposed to be random variables that obey to some probability distribution[6]. Further, if they are input of transductions the uncertainty is propagated throughout the net, and we can no longer refer to a specific solution of the system and we must talk about the probability of getting that solution. Thus, in order to reason about the behavior of the system, it turns out that it is not helpful to consider single solutions because we can get more interesting insights if we look at the statistics of the distribution (of solutions); for example we can use its expected value. The following example, adapted from (St-Aubin 2005), makes this last point clearer.

Example 2.5 (The effect of randomness on fixpoints) *Let's consider the following dynamical systems:*

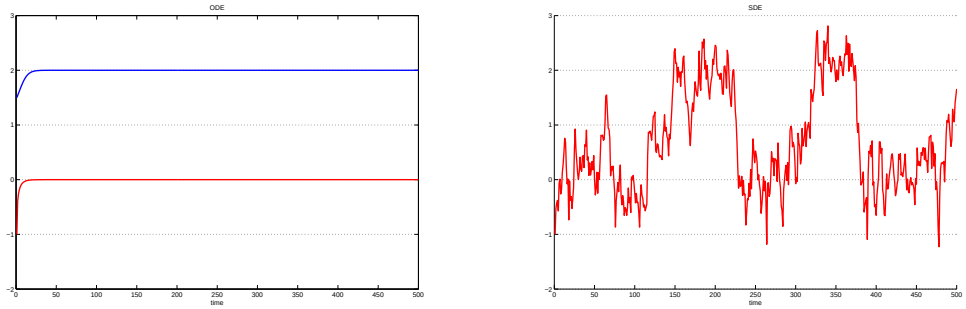
$$\dot{X}_t = -X_t(X_t - 1)(X_t - 2) \tag{2.8}$$

$$\dot{X}_t = -X_t(X_t - 1)(X_t - 2) + N_t; \tag{2.9}$$

where eq. 2.8 is a deterministic system with three equilibria: 0 *and* 2 *(stable attractors) and* 1 *(unstable). Its behavior is fully determined by its initial value and it reaches one of the two stable fixpoints based on this initial value. For examples, figure 2.8(a) shows the solution of equation 2.8 for two distinct initial values: one in a neighborhood of* 0 *and the other in a neighborhood of* 2*: the two attractors are reached quite soon and the solution doesn't change anymore.*

The second system (eq. 2.9) is stochastically affected by a simple Brownian motion process. Figure 2.8(b) shows a sample path for system 2.9, with an initial value of $X = 0$*. As a consequence of the Brownian motion perturbation, the system fluctuates around this attrac-*

[6] Recall that random variables in a PCN are those locations that are the output of a generators.



(a) ODE described by equation 2.8 (b) SDE described by equation 2.9

Figure 2.8: Differences between ordinary and stochastic systems.

tor. It can happen that a large enough noise disturbance pushes the system over the value of 1 *– i.e. it leaves the region of attraction – causing the system to be attracted toward the other equilibrium, at* $X = 2$*. Another spike of noise can flip the system back to the lower equilibrium and so on (see fig. 2.8(b)). This example shows the effect of uncertainty on the system and its behavior. In this case, we can no longer refer to any fixpoint.*

However, the system will reach a stationary distribution. That is, in the long run, the probability distribution of the system will remain unchanged, independent of time. The empirical distribution corresponding to a sample path is showed in figure 2.5. One can clearly observe that the system is symmetrically distributed with higher weight around the two stable equilibria located at $X = 0$ *and* $X = 2$*.*

Example 2.6 (Dependence of the fixpoint on the actual run of the system) *This second example is a slight modification of the previous one and it shows that it it not safe to look at a single trace of the system because the possibly well–defined attractor of a single trace could lead to misleading conclusion about the overall behavior of the system.*

Let's consider the following system of equations, which describe the behavior of a fully–interconnected system of two neuron–like *computational units*[7].

[7]See chapter 4 for more details about artificial neuron–based systems and their relationships with the PCN frameworks.

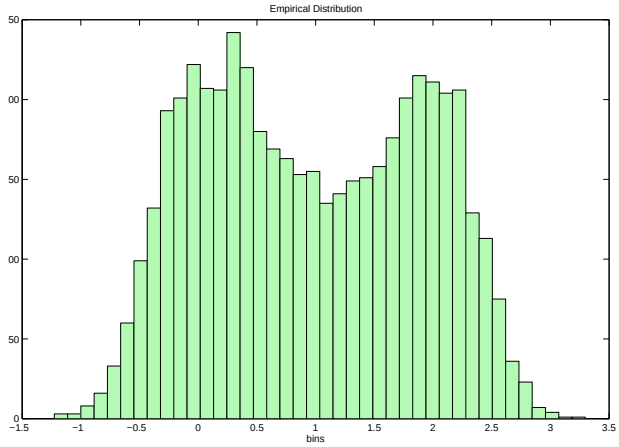


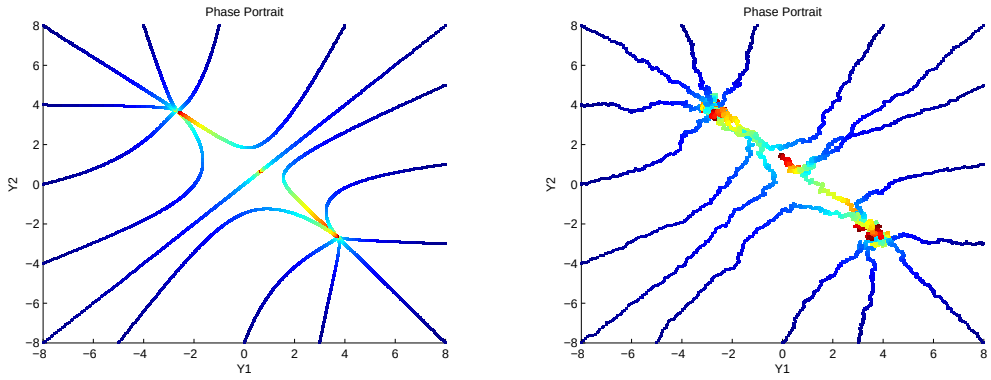
Figure 2.9: Stationary, empirical distribution of a sample path for system described by equation 2.9

$$\dot{y}_1 = -y_1 + w_{11}\sigma(y_1 - \theta_1) + w_{12}\sigma(y_2 - \theta_2)$$
$$\dot{y}_2 = -y_2 + w_{12}\sigma(y_1 - \theta_1) + w_{22}\sigma(y_2 - \theta_2) \tag{2.10}$$

where y_i are the variables, w_{ij} are constant weights in the equations and σ is the standard sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$.

The trajectories of the system will depend on the initial state x_0. Figure 2.10(a) shows some representative trajectories corresponding to the parameter values $w_{11} = w_{22} = 4, w_{12} = w21 = -3, \theta_1 = \theta_2 = 0$. The system exhibits two stable equilibrium points near $(-3, 4)$ and $(4, -3)$ with basins of attraction that lie respectively on the top–left and bottom–right of the diagonal of the reference system. Given that the initial state is in one region or in the other of the plan, each solution will reach the corresponding attractor. Thus the behavior of the system is fully determined by its initial condition.

Figure 2.10(b) shows what happens to the system if we perturbate it with a simple Brow-



(a) ODE described by equation 2.10

(b) SDE described by equation 2.10 stochastically affected by a simple Brownian motion process

Figure 2.10: Stochastic Perturbations of a Dynamical System.

nian motion process. Differently from the previous example, here the two basins of attraction are deeper *enough and the stochastic perturbation doesn't influence the single traces (see figure 2.10(b)). Even if the system starts from the origin – i.e. on the edge between the two basins – the trajectory won't fluctuate back and forth between the two attractors. However, this doesn't mean we can rely on the two attractors to define the fixpoint of the distributions. In fact, figure 2.6 shows what happens when we let the system evolve several times with the same initial conditions. Again we obtain an empirical distribution that is more insightful that any single path.*

It is clear from the above example that we can get more insightful information if we look at the distribution of the solutions instead of at one single solution whose measure is null[8].

Now that we are equipped with the idea behind the notion of semantics of a PCN, it is easy to understand the following formal definitions.

Let's consider a signature $\Sigma = \langle S, F \rangle$ with a special sort $c \in S$ defined to represent clocks. We say that Σ is the signature of a $PCN = \langle Lc, Td, CN \rangle$ and we write PCN_Σ, if:

[8]Here the expression *measure of a solution* means probability of occurrence of the event associated to the single solution in the sample space with all the possible solution. Hence, it is obvious that the probability associated to the single event is zero.

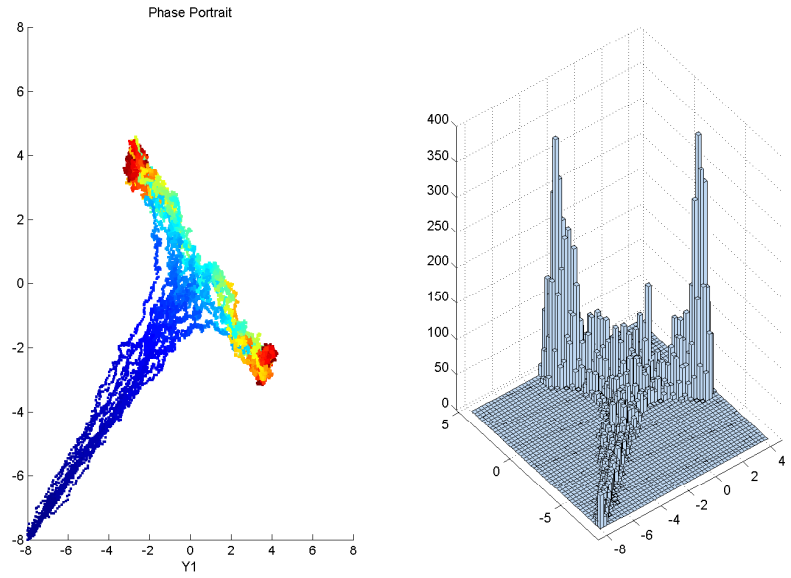


Figure 2.11: Stationary, empirical distribution of several sample paths for system described by equation 2.10 stochastically affected by a simple Brownian motion process

– each location $l \in Lc$ is associated with a sort $s \in S$ (we write s_l to refer to the sort of l);

– for each transduction $f \in Tc$ the sorts of its input and output ports are as follows:

1. if f is a transliteration of a function (i.e. $f : s^* \rightarrow s$) of F, the sort of the output port is s and the sort of the input port i is $s^*(i)$;
2. if f is a unit delay δ^s or a transport delay Δ^s, the sort of both input and output port is s;
3. if f is an event–driven transduction, the sort of the event input port is c, the sort sort of the other ports are the same as its primitive transduction;

Definition 2.10 (Semantics of *PCNs*) *The semantic of a probabilistic constraint net PCN on a dynamics structure $\langle \mathcal{V}, \mathcal{F} \rangle$, denoted $[\![PCN]\!]$, is the least stationary distribution of the set of equations $\{o = F_o(\mathbf{x})\}_{o \in O(PCN)}$. Moreover, if F_o is a continuous or pathwise continuous trans-*

duction in $\mathcal{F}$ for all $o \in O(PCN)$; then $[\![PCN]\!]$ is a continuous or pathwise continuous transduction from the input trace space to the output trace space, i.e. $[\![PCN]\!] : \times_{I(PCN)} A_{s_i}^{\Omega \times \mathcal{T}} \rightarrow \times_{O(PCN)} A_{s_o}^{\Omega \times \mathcal{T}}$. ■

Of course, it is possible to define the semantics of the modules (see St-Aubin (2005)) and it can be shown that the combination operations defined in the previous section do preserve the semantics, thus we are allowed to build complex systems by means of hierarchical composition of simpler modules.

Chapter 3

Behavioral Verification of Robotic Systems

In this chapter I summarize the notion of average–timed ∀–automata and discuss its links with behavioral specification and verification within the PCNs framework.

The key idea behind the chapter is that the online satisfaction of the local constraints[1] imposed on the dynamics of a robotic system does not guarantee that the robot will satisfy any global behavioral constraint, such as – for example – "to accomplish a task correctly within a limited amount of time". Unfortunately, PCNs are not tailored for representing global constraints on the behavior of a systems. Even if, at least in principle they are expressive enough to formalize behavioral requirements, it is not reasonable to do it in any real application.

The approach proposed within the PCNs framework to overcome this problem is to define a different, automata–based specification language by means of which we can easily formulate behavioral constraints. Such an approach has a second, major advantage since it allow us to design and (possibly) implement formal verification procedures. To design (semi)automatic procedures is thus the ultimate goal of researchers in this area.

Even is my personal contribution to the development of specification and verification methods has been quite limited, I believe this chapter is conceptually fundamental for the understanding of some following topics discussed in this thesis. Moreover, as discussed in the next chapter, one of the ongoing developments of PCNJ is to provide the user with the

[1] Which can be easily described by means of PCNs.

possibility of defining average timed ∀–automata within PCNJ.

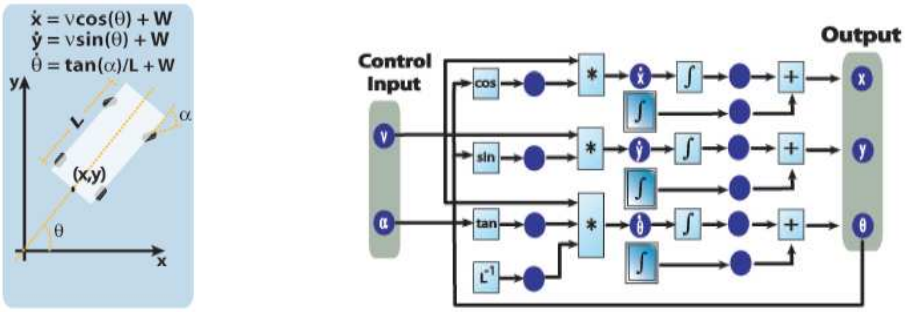
The chapter is organized as follows. Section 3.1 provides some preliminary concepts of behavioral specification and verification and describes the meaning of formal behavior for a robotic system. Sections 3.2 and 3.3 describe the to specification languages and their links. Finally, section 3.4 presents the model checking approach proposed by St–Aubin for the verification of any stochastic, hybrid dynamical system.

3.1 Concepts of Behavioural Specification and Verification in Robotics

Since modern society is increasingly dependent on complex software (and hardware) systems for managing and processing sensitive and critical information, the consequences of failures can become extremely severe. Hence, computer scientists have been developing formal methods for decades in order to model the behavior of software systems and to verify that these models satisfy some desired properties. Nowadays formal specification and verification of software is an essential stage in many areas of software engineering, and it is a topic any computer science graduate student is – or should be – familiar with. However, in contrast to what happens for software system development, the use of formal methods is not prevalent in almost any research areas of Robotics.

This section provide a *smoother transition* from concepts that are known and well assessed in the software specification and verification research community to their new, specific use in Robotics.

Let's start from the fundamental difference between *system modeling* and the *specification of behavioural constraints*; we assumed the existence of this difference throughout chapters 1 and 2, yet it might still be unclear. Although the two concepts might appear similar, they are very different. In short, the modeling task focuses on the dynamics of the system and how different components interact among each other, i.e. it imposes local constraints on the



(a) System Dynamic (b) PCN model

Figure 3.1: PCN model of the dynamics of a system comprised of a mobile car with uncertain actuators moving in a 2D environment. [Taken from**?**]

systems dynamics. On the other hand, the specifications of a system impose global constraints on its evolution/behaviours. For example, the dynamics of a (car–like) mobile robot can be modelled by differential equations following basic laws of physics such as the relation between velocity and acceleration – e.g. $\dot{x} = v\cos(\theta) + W_t^x, \dot{y} = v\sin(\theta) + W_t^y, \dot{\theta} = v/R + W_t^\theta$; which are the laws governing the system depicted in figure 3.1(a)[2]. These laws represent the constraints on the dynamics. However, although these represent well the local behaviour of the system, it does not preclude the robot from hitting people as it is roaming around. If the goal of the robot is to deliver something (e.g. a coffee) somewhere, then we might be able to represent our wish that the robot will always be successful when attempting to deliver the coffee. Such restrictions are global constraints on the behaviours of the system and cannot be represented easily with PCNs only. They can, however, be compactly expressed with a ∀–automaton specification as it is shown in Figure 3.2.

Once one is equipped with a model of the dynamics of a system and with requirement specifications on the global behavior of the system, a key question to ask is whether the behavior of the system satisfies these requirements. This is called behavioural verification

[2]Figure 3.1(b) shows the corresponding PCN.

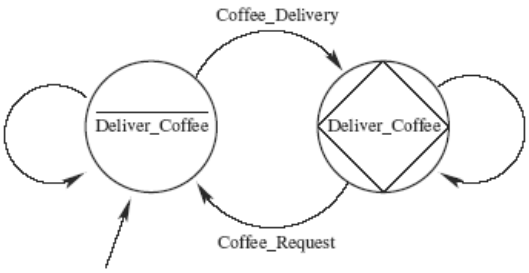


Figure 3.2: Robot Delivery ∀–Automaton Specification

and is the topic addressed in sections 3.4.

3.2 ∀–Automata

∀–automata are non–deterministic finite state automata over infinite sequences. They were originally proposed to specify requirements and temporal properties of concurrent programs (Manna and Pnueli 1987) or time traces from deterministic dynamical systems Zhang (1994), Zhang and Mackworth (1996).

Formally, a ∀–automaton is defined as follows.

Definition 3.1 (Syntax of ∀–automata) *A* ∀–automaton $\mathcal{A}$ *is a quintuple* $\langle Q; R; S; e; c\rangle$ *where* Q *is a finite set of automaton states,* $R \subseteq Q$ *is a set of recurrent states and* $S \subseteq Q$ *is a set of stable states. With each* $q \in Q$*, we associate an assertion* $e(q)$*, which characterizes the entry condition under which the automaton may start its activity in* q*. With each pair* $q, q' \in Q$*, we associate an assertion* $c(q, q')$*, which characterizes the transition condition under which the automaton may move from* q *to* q'*.* ■

R and S are generalizations of *accepting* states to the case of infinite inputs. All the

other states of the automaton, i.e. $B = Q - (R \cup S)$, are called *bad* states because they are nonŋ-accepting states.

A ∀–automaton is called complete iff the following requirements are met:

- $\bigvee_{q \in Q} e(q)$ is valid.

- $\forall q \in Q, \bigvee_{q' \in Q} c(q, q')$ is valid.

Any automaton can be transformed to a complete automaton by introducing an additional bad (error) state q_E, with entry condition $e(q_E) = \neg\left(\bigvee_{q \in Q} e(q)\right)$, and the transition conditions:

$$\begin{aligned} c(e_E, q_E) &= true \\ c(q_E, q) &= false \quad \text{for each} \quad q \in Q \\ c(q, q_E) &= \neg(\bigvee_{q' Q} c(q, q')) \quad \text{for each} \quad q \in Q \end{aligned}$$

Like any kind of automaton, it is possible to introduce a useful, simple graphical representation for ∀–automata. Let's consider a labelled, directed graph whose nodes represents automaton–states and whose arcs are transition relations. We say that such a graph is a representation of a ∀–automaton iff:

1. for each automaton state there exists one node of the graph;

2. each initial automaton state[3] is marked by a small arrow () pointing to it;

3. arcs, drawn as arrows, connect some pairs of automaton states;

4. each recurrent state is depicted by a diamond inscribed within a circle;

5. each stable state is depicted by a square inscribed within a circle;

[3]Initial state are those for which there exists an entry assertion $e(q) \neq false$.

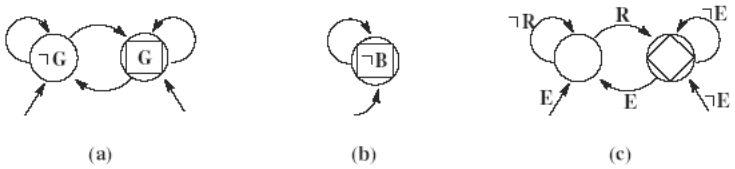


Figure 3.3: Examples of ∀–Automata: (*a*) goal achievement, (*b*) safety, and (*c*) bounded response.

6. nodes and arcs are labeled by assertions.

Labels, attached to nodes and arcs, define the entry conditions and the transition conditions of the associated automaton as follows:

- Let $q \in Q$ be a node in the diagram corresponding to an initial automaton–state. If q is labeled by ψ and the entry arc is labeled by ϕ, the entry condition $e(q)$ is given by $e(q) = \psi \wedge \phi$. If there is no entry arc, $e(q) = false$.

- Let $q, q\prime$ be two nodes in the diagram corresponding to automaton–states. If $q\prime$ is labeled by ψ, and arcs from q to $q\prime$ are labeled by $\phi_i, i = 1, \ldots, n$, the transition condition $c(q, q\prime)$ is given by $c(q, q\prime) = (\phi_1 \vee \ldots \vee \phi_n) \wedge \psi$. If there is no arc from q to $q\prime$, $c(q, q\prime) = false$.

A diagram representing an incomplete automaton can be interpreted as a complete automaton by introducing an error state and associated entry and transition conditions. Some examples of ∀–automata are shown in figure 3.2

The formal semantics of discrete ∀–automata is defined as follows. Let A be a domain of values. An assertion α on A corresponds to a subset $V(\alpha) \subseteq A$. A value $a \in A$ satisfies an assertion α on A, written $a \models \alpha$ or $\alpha(a)$, iff $a \in V(\alpha)$. Let $\mathcal{T}$ be a discrete time structure and $v : \mathcal{T} \rightarrow A$ be a trace. A *run* of $\mathcal{A}$ over v is a mapping $r : \mathcal{T} \rightarrow Q$ such that (1) $v(0) \models e(r(0))$;

and (2) for all $t > 0$, $v(t) \models c(r(pre(t)), r(t))$. A complete automaton guarantees that any discrete trace has a run over it.

If r is a run, let $Inf(r)$ be the set of automaton states appearing infinitely many times in r, i.e., $Inf(r) = \{q | \forall t \exists t_0 \geq t, r(t_0) = q\}$. Notice that the same definition can be used for continuous as well as discrete time traces. A run r is defined to be accepting iff:

1. $Inf(r) \bigcap R \neq \emptyset$, i.e., some of the states appearing infinitely many times in r belong to R, or

2. $Inf(r) \subseteq S$, i.e., all the states appearing infinitely many times in r belong to S.

We can now introduce the definition of formal semantics for ∀–automata.

Definition 3.2 (Semantics of ∀–automata) *A ∀–automaton $\mathcal{A}$ accepts a trace v, written $v \models \mathcal{A}$, iff all possible runs of $\mathcal{A}$ over v are accepting.* ■

Figure 3.2 shows three different ∀–automata whose semantics are as follows. (*a*) accepts the traces of a system which eventually will always satisfy the goal condition *G*; (*b*) accepts the traces of a system that should never satisfy the unsafe condition *B.* (*c*) accepts the traces of a system that satisfy a bounded response constraint, i.e. whenever event *E* occurs, the response *R* will occur in bounded time.

One should note that the proposed definition of semantics differs in the way it handles non–determinism from the semantics of conventional automata. A conventional automata *C*, which could, in this context, also be called a ∃–automata, accepts a language if there exists at least one run over *C* which is accepting. However, in the context of behavior verification, having at least one run satisfying the requirements is obviously not a strong enough statement as in the case of a safety requirement, this is generally not what should be defined as a safe system. Moreover, for deterministic systems, which are defined completely by a single trace,

it is meaningful to require the trace be accepted. However, when modeling a stochastic system, asking for all traces to be accepted (which we referred to as satisfying the requirements at level $\alpha = 1$) might be too demanding. Indeed, there might be a very small probability that the system will move into a set of absorbing bad states, hence never satisfying the behavioural constraints. However, if this probability (which is equivalent to the measure of all sample traces leading to the absorbing bad states) is small enough so that these events rarely occur, one might be willing to accept the risk to work with a system which satisfy the requirements at a level α where $\beta < \alpha < 1$ and β is the safety threshold.

Before moving on, it might be helpful to discuss the notion of verification at level $\alpha = 1$ of a stochastic dynamical systems. What type of restrictions on the system itself does this create? Intuitively, perfect satisfaction of a set of behavioural constraints amounts to the system not possessing any absorbing bad states. By absorbing we refer to the case where the system enters this bad state and never leaves it. In practice, for a system to not possess any absorbing bad states requires that for any state of the stochastic dynamical system associated with a bad automaton state, there must exist a path with positive probability which leads to an accepting state (associated to either R or S). Indeed, for a large class of systems with absorbing bad states, these states corresponds to a situation where the robotic agent is down in one way or another. Hence, repair or restart would be needed to ensure that the system can continue operating. One could take this *repair* into account and modify the state space so that once the agent arrives to a absorbing bad state, a transition occurs with probability one which relocates the agent to *restart* state. This simple modification removes absorbing bad states and thus allows the verification method to be applied to a vast class of systems.

3.3 Average–Timed ∀–Automata

The class of constraints that we can express by means of simple ∀–automata doesn't contain the fundamental subclass of those behavioural constraints that encompass explicitly temporal specifications. For example, in robotics it is quite meaningless to have a formal guarantee that the robot – whenever a significant event E occurs – will produce a response R in bounded time. Actually, we are more interested in proving that the response R will occur in a limited time, that is to say we want to attach a finite constant k to the former time bound.

Timed ∀–automata were originally proposed by Zhang (1994) in order to augment basic ∀–automata with timed automaton states and time bounds. Both (Zhang and Mackworth 1996) and(Mackworth and Zhang 2003) provide the formal definition of this family of automata and a description of the their properties. They are very useful for further references.

Unfortunately, the approach based on timed ∀–automata is not well suited for stochastic dynamical systems. In fact, in the stochastic case, it is not possible any more to talk about satisfying a given time constraint in an absolute way but rather we might accept a kind of *on–average* satisfaction, i.e. we can beyond the concept of time constraint and define that of *average* time constraint. The idea behind average time constraint is that although we cannot prove that a stochastic dynamical system can always satisfy some given time constraint, we can show that the average behavior of the system does satisfy the constraints. All these informal considerations can be defined formally.

Definition 3.3 (Syntax of average–timed ∀–automata) *An* average–timed ∀–automaton $\mathcal{AT}\mathcal{A}$ *is a triple* $\langle \mathcal{A}, T, \tau \rangle$ *where* $\mathcal{A} = \langle Q, R, S, e, c \rangle$ *is a ∀–automaton,* $T \subseteq Q$ *is a set of average–timed automaton states and* $\tau : T \cup \{bad\} \rightarrow \mathbb{R}^+ \cup \{\infty\}$ *is an average–timing function.*

Of course, any ∀–automaton is equivalent to a special average–timed ∀–automaton – the one obtained by setting $T = \emptyset$ and $\tau(bad) = \infty$. We attach a nonnegative real number to any T–state, indicating its average–time bound.

As anticipated earlier, we cannot define the acceptance of a single trace by an average–timed ∀–automata. Since we are no longer interested in the behavior of individual traces[4] we might better consider the behavior of a set of traces. Therefore, expected time constraints should be satisfied by the average behaviours of this ensemble of traces.

Let $\mathcal{B}$ be the considered behavior. We define a *run* r of $\mathcal{ATA}$ over $\mathcal{B}$ has being a *run* of $\mathcal{A}$ over every trace $v : \mathcal{T} \rightarrow A$ in the behavior $\mathcal{B}$. A *run* r is accepting for $\mathcal{ATA}$ iff:

1. r is accepting for $\mathcal{A}$, and

2. r satisfies the expected time constraints. Let's consider a time interval $I \subseteq \mathcal{T}$ and a segment q^* of $r - q^* : I \rightarrow Q$ and $q^* = r_{|I}$ – whose measure is denoted by $\mu(q^*)$. Furthermore, let $\mu_B(q^*)$ denote the measure of bad automaton states in q^*, and $Sg(q)$ be the set of segments of consecutive q state in r. Finally, let BS be the set of segments of consecutive B and S–states in r, i.e., $q^* \in BS$ implies $\forall t \in I, q^*(t) \in B \cup S$.

   The expected time constraints for r can be formulated as:

   (a) (local time constraint) $\forall q \in T, q^* \in Sg(q), \mathbb{E}(\mu(q^*)) \leq \tau(q)$ and

   (b) (global time constraint) $\forall q^* \in BS, \mathbb{E}(\mu_B(q^*)) \leq \tau(bad)$.

   where $\mathbb{E}(\cdot)$ denotes obviously the expectation over all traces v of $\mathcal{B}$.

It is now possible to state the final definition of semantics of an *average–timed ∀–automaton*.

Definition 3.4 (Semantics of average–timed ∀–automata) *An average–timed ∀–automaton $\mathcal{ATA}$ accepts a set of traces $\mathcal{B}$, written $\mathcal{B} \models \mathcal{ATA}$, iff all possible expected runs of $\mathcal{ATA}$ over $\mathcal{B}$ are accepting.*

[4] Because each individual trace is a null–measure event.

3.4 Model–Checking Approach for Behavioral Verification

In this section I present the model–checking approach for behavioural constraints verification proposed by St-Aubin (2005), whose thesis work showed the existence of well defined *behavioural constraint verification rules* for both arbitrary time and domain structures. These general rules are essential to provide an understanding of general behaviours of stochastic hybrid dynamical systems. However, St–Aubin provided a (semi) automatic verification method only for one special case of finite domain PCNs and discrete time. At the present time, thus, no algorithm either automatic or semi–automatic has been developed yet for behavior verification of general stochastic hybrid dynamical systems. If we restrict ourselves to consider non–probabilistic systems only, then the we can rely on the results obtained in Zhang (1994), Zhang and Mackworth (1996) that showed a further nice property of the resulting verification algorithms, i.e. they are polynomial in both the size of the model and the size of the specification.

Once again I recall that the focus of the present thesis is not on the theoretical aspects of behavioral verification for general hybrid systems. Instead, I aim at providing convincing arguments in favor of using PCNs framework in the robotic research area. From this point of view I can dodge the description of the verification rules for the general case and focus on those for discrete–time finite–domain stochastic systems, which – fortunately – turn out to be a suitable level of description for most of the real robotic systems. This last claim may seem a bit contradictory with respect to the initial claims about generality of PCNs framework. In a way I must agree with these concerns and admit that this part of the framework still needs significant improvements. However, the existence of verification rules for the general case is a first big step towards the goal of either finding an algorithm for the general case or proving the non–existence of such an algorithm.

Let's now introduce the verification rules. This method applies to any stochastic state

transition system $\mathcal{S}_{\mathcal{B}} = \langle S_{\mathcal{B}}, \mathbb{P}, \rangle$ associated to a time–invariant Markovian behavior $\mathcal{B}$ in discrete time. Let's denote $s \rightsquigarrow s'$ an allowed transition from state s to state s' of the system – i.e. there is a non-zero probability of transition. Also $\{\varphi\}\mathcal{B}\{\psi\}$ denotes the condition: $\varphi(s) \wedge (s \rightsquigarrow s') \rightarrow \psi(s')$ is valid. We call $\mathcal{ATA}$ the the average timed $\forall$–automaton $\langle \mathcal{A}, T, \tau \rangle$ representing the behavioural constraints for the stochastic dynamical system $\mathcal{S}_{\mathcal{B}}$.

The verification method comprises three basic types of rules:

Invariance rules (I) . A set of propositions $\{\alpha_q\}_{q \in Q}$ is a set of *invariants* for the behavior $\mathcal{B}$ and specification $\mathcal{ATA}$ iff:

1. *Initiality*: $\forall q \in Q, \Theta \wedge e(q) \rightarrow \alpha_q$, and
2. *Consecution*: $\forall q, q' \in Q, \{\alpha_q\}\mathcal{B}\{c(q, q') \rightarrow \alpha_{q'}\}$

It is possible to show that, given a set of invariants for an automata $\mathcal{ATA}$ and a behavior $\mathcal{B}$, any trace in $\mathcal{B}$ always brings from one state that satisfy the invariant conditions to a destination state that still satisfy the invariant conditions; no matter which (possibly uncertain) transition occurs.

Stability (Lyapunov–based) rules (S) A set of partial functions $\{\rho_q\}_{q \in Q} - \rho_q : \mathcal{S}_{\mathcal{B}} \rightarrow \mathbb{R}^+ -$ is called a set of *Lyapunov functions* for $\mathcal{ATAT}$ and $\mathcal{B}$ iff they satisfy the following conditions:

1. *Definedness*: $\forall q \in Q, \alpha_q \rightarrow \exists w \in \mathbb{R}^+, \rho_q = w$.
2. *Non–increase*: $\forall q \in S, q' \in Q, \{\alpha_q \wedge \rho_q = w\}\mathcal{B}\{c(q, q' \rightarrow \mathbb{E}(\rho_{q'}) \leq w\}$.
3. *Decrease*: $\exists \epsilon > 0, \forall q \in B, \exists q' \in Q, \{\alpha_q \wedge \rho_q = w\}\mathcal{B}\{c(q, q' \rightarrow \rho_{q'} - w \leq -\epsilon\}$.

Condition $(S2)$ requires that for each stable state $q \in S$, the transitions from q lead on average to a state for which the value of the Lyapunov function is less than or equal to the current value. Condition $(S3)$ requires that for each bad state $q \in B$ there exists

at least one allowed transition (i.e., with positive probability) leading to a state with strictly smaller Lyapunov value. This is a formal requirement that can only be satisfied if there are no absorbing bad states in the system under study.

Average Timeliness rules (AT) Let $\mathcal{AT}\mathcal{A} = \langle \mathcal{A}, T\tau \rangle$ be an average–timed $\forall$–automata. Assume, without loss of generality, that time is encoded in the stochastic state transition system. Let's define $\lambda : \mathcal{S}_{\mathcal{B}} \rightarrow \mathcal{T}$ as a function of time measure on states returning the time until the next transition.[5]. Let's introduce two different types of timing functions, associated with the local and global average–time bounds respectively.

A set of partial functions $\{\gamma_q\}_{q \in T}$ is called a set of *local timing functions* for $\mathcal{B}$ and $\mathcal{AT}\mathcal{A}$ iff $\gamma_q : \mathcal{S}_{\mathcal{B}} \rightarrow \mathbb{R}^+$ satisfies the following conditions:

(L1) *Boundedness*: $\forall q \in T, \alpha_q \rightarrow \lambda \leq \gamma_q \leq \tau(q)$.

(L2) *Decrease*:$\forall q \in T, \{\alpha_q \wedge \gamma_q = w \wedge \mathbb{E}(\lambda) = l\}\mathbb{B}\{c(q, q' \rightarrow \mathbb{E}(\gamma_q) - w \leq -l\}$.

A set of partial functions $\{\eta_q\}_{q \in Q}$ is called a set of *global timing functions* for $\mathcal{B}$ and $\mathcal{AT}\mathcal{A}$ iff $\eta_q : \mathcal{S}_{\mathcal{B}} \rightarrow \mathbb{R}^+$ satisfies the following conditions:

(G1) *Definedness*: $\forall q \in Q, \alpha_q \rightarrow \exists w \in \mathbb{R}^+, \eta_q = w$.

(G2) *Boundedness*: $\forall q \in B, \alpha_q \rightarrow \eta_q \leq \tau(bad)$.

(G3) *Non–increase*: $\forall q \in S, q' \in Q, \{\alpha_q \wedge \eta_q = w\}\mathcal{B}\{c(q, q' \rightarrow \mathbb{E}(\eta_{q'}) \leq w\}$.

(G4) *Decrease*: $\forall q \in S, q' \in Q, \{\alpha_q \wedge \eta_q = w \wedge \mathbb{E}(\lambda) = l\}\mathcal{B}\{c(q, q' \rightarrow \mathbb{E}(\eta_{q'}) \leq w\}$.

The above *verification rules* – i.e. Invariance (I), Stability (S) and Average-Timeliness (AT) – can be used with a behavior $\mathcal{B}$ and an average–timed automaton $\mathcal{AT}\mathcal{A} = \langle \mathcal{A}, T, \tau \rangle$ as follows:

(I) Associate with each automaton state $q \in Q$ a state formula α_q, such that $\{\alpha_q\}_{q \in Q}$ is a set of invariants for $\mathcal{B}$ and $\mathcal{A}$.

[5]For the special case of discrete time systems on $\mathbb{N}$, $\lambda = 1$ uniformly.

(S) Associate with each automaton state $q \in Q$ a partial function ρ_q, such that $\{\rho_q\}_{q \in Q}$ is a set of Lyapunov functions for $\mathcal{B}$ and $\mathcal{A}$.

(AT) Associate with each average–timed automaton state $q \in T$ a partial function γ_q, such that $\{\gamma_q\}_{q \in T}$ is a set of local timing functions for $\mathcal{B}$ and $\mathcal{AT}\mathcal{A}$. Associate with each automaton state $q \in Q$ a partial function η_q, such that $\{\eta_q\}_{q \in Q}$ is a set of global timing functions for $\mathcal{B}$ and $\mathcal{AT}\mathcal{A}$.

The final step it to state the main result related to these verification rules: St–Aubin demonstrated that, if we are equipped with a set of invariants, Lyapunov functions and local and global timing functions, then the behavioural verification is sound and complete. This is stated by St-Aubin (2005) as Theorem 7.2, which is reproduced here for completeness.

Theorem 3.1 (Reproduced from St-Aubin (2005), Theorem 7.2 (Verification Rules)) *For any state-based and time–invariant behavior $\mathcal{B}$ with an infinite time structure and a complete average–timed $\forall$–automaton $\mathcal{AT}\mathcal{A}$, the verification rules are sound and complete, i.e., $\mathcal{B} \models \mathcal{AT}\mathcal{A}$ iff there exist a set of invariants, Lyapunov functions and timing functions.*

Chapter 4

Models Subsumed by PCN

PCNs framework is a non trivial extension of CNs framework because it allows for the modelling of both systems with uncertainty and systems which behave probabilistically. This is a valuable asset because now we can model a number of systems – by means of PCNs – that couldn't be represented as CNs. This is a non–trivial claim that deserves to be proved more formally, and this chapter contains a number of positive results that provide good evidence for such a claim.

I look more carefully into the relationships between PCNs and a several deterministic/probabilistic models and algorithms commonly used in Robotics. I show that they are special cases of PCNs by providing – for each model/algorithm – the PCN that computes exactly the same thing, i.e., the proposed PCN preserves the semantics of the computation. Since many of these models are widespread in Robotics and Computer Science as well, we reap two main benefits from the results described in this chapter. On one side they further demonstrate the flexibility of PCNs framework from the point of view of both theoretical and practical expressivity. On the other side they are an effective contribution to the general discussion developed throughout this thesis about the use of PCNs within robotic research areas.

Furthermore, in this chapter I focus mostly on learning–related models and algorithms; this is because learning is becoming increasingly an effective tool to build fundamental modules of mobile robots. Once we are equipped with PCNs that implement learning algorithms we might start asking what kind of benefit (if any) we can get from the formal specification

and verification capabilities provided by average timed ∀–automata within PCNs framework. I do say something about this interesting topic in the last chapter of this thesis.

The chapter is organized as follows: in section 4.1 I motivate the usefulness of this chapter by explaining what we actually gain by expressing another computational model as a PCN. The following sections are devoted to showing the equivalence with PCNs of Artificial Neural Networks (sec. 4.2), Continuous Time Recurrent Neural Networks (sec. 4.3), Markov Models (sec. 4.4), Reinforcement Learning and Markov Decision Processes (sec. 4.5), and finally Kalman Filters (sec. 4.6).

4.1 A Few Preliminary Remarks on the Computational Power of PCNs

This chapter in general and this section in particular discuss the *computational expressive power* of the PCNs framework.

Since Zhang (1994) showed that CNs are expressive enough to compute any partial recursive function and thus that CNs are universal computing devices[1], some could argue that the present chapter is a bit redundant. Hence some preliminary remarks are necessary in order to clarify the usefulness of what follows.

The notions of *expressive power* or *expressiveness* of knowledge representation languages have been investigated by most papers on knowledge representation for nearly a decade (e.g. Woods (1983), Levesque and Brachman (1987), Nebel (1990)). Surely, we may have an intuitive idea of what these terms mean, but indeed several formal definitions of expressiveness have been already proposed – (Baader 1996), to cite just one example – on which there is as common agreement among researchers in theoretical computer science. Following these definitions, we can to point out the existence of two distinct criteria for evaluating how expressive

[1] In the sense of Turing computability, i.e. form the point of view of Theoretical Computer Science

a formal language is (Tesfagiorgis 2006):

1. *Theoretical expressivity*: One language is said to be (theoretically) more expressive than another language, if whatever the latter can express can also be expressed by the former too, while the reverse is not necessarily true.

2. *Practical expressivity*: deals with the ease and naturalness of a language in expressing real–life systems. For instance, a Turing machine is theoretically as expressive as any programming language, however writing a usable program in a Turing machine language is far more complex than in a programming language.

These criteria tell us two things: (1) from a theoretical point of view, if we prove that PCNs are expressive enough to represent – let's say – any (probabilistic) Turing machine, then at least in principle researchers could use the framework whenever they need because there will certainly exist the PCN suitable for them. (2) From a practical point of view, if we demonstrate that a specific algorithm can be expressed as a PCN by means of a constructive proof, then we provide researchers with a method for building that PCN. A further immediate reward from this latter approach is the possibility to discuss directly the efficiency of one specific PNC module and compare it with other different implementations of the same algorithm.

In the next sections I'll be seeking constructive proofs for specific algorithms – i.e. I'll be adhering to the practical perspective.

4.2 Neural Networks and PCNs

In this section I build a bridge between PCNs framework and one of the most popular approach to machine learning, i.e. the one based on artificial neural networks (ANNs) that gained increasing popularity over the last decades. ANNs are very often used as part of robotic architectures ([...]) and thus it is a valuable contribution indeed to show their relationship with PCNs.

An ANN is an interconnected group of computational units that are called *artificial neurons*. Of course, the term "neural network" suggests biological systems, yet the biological roots of ANNs are irrelevant to the present discussion. From the point of view of artificial intelligence, ANNs are essentially simple mathematical models defining a function and they are extremely useful when we do not know explicitly the expression of such a function.

The ability to *learn* is surely the most interesting one for ANNs and it triggered interesting debates among researchers during the early years of ANNs about the *ontological* meaning of learning machines. Nevertheless learning in ANNs is based on a few simple mathematical considerations that can be summarized as follows. Informally, ANNs can be considered adaptive systems that change their structure based on the flow of information that passes through the network. More formally, ANNs are families of function approximators: they are mathematical models with a sufficiently large number of parameters by means of which it is possible to define any function , given that a set of input–target training examples is provided to the ANN. Of course, I'm referring here to the *supervised approach* to neural networks. In this approach, in order to *learn* something, we need to collect pairs of input and corresponding target, the targets being a kind of teacher's specification of what the neural network's response to that input should be. Finally, we can summarize these consideration by stating that learning in ANNs takes place owing to their structure and to suitable learning rules defined over the structure. The learning rules specify the way in which the neural network's parameters change over time.

In this section the focus is on a subclass of ANNs, the so called Multi–layer feedforward neural networks (MLFFNNs). Most of the learning algorithms commonly used to train MLFFNNs is based on the technique of on–line gradient–descent and is called back–propagation. I will show that, for every ANN, there exist a well–defined PCN; moreover, the back–propagation algorithm itself can be described as a specific PCN.

Some formal definitions are required preliminary to the description of ANNs as PCNs.

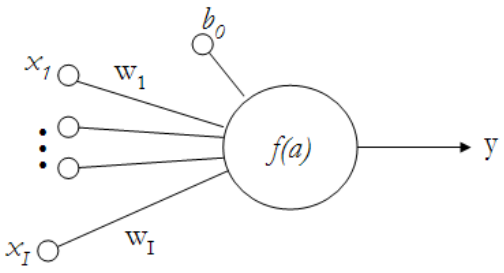


Figure 4.1: Schematic representation of a single artificial neuron.

Let's start from the basic computational unit of an ANN.

Definition 4.1 (Artificial Neuron) *A single artificial neuron is a feedforward, computational device that has a number I of inputs x_i and one output y (fig. 4.1). Associated with each input is a weight $w_i (i = 1, \ldots, I)$. There usually is an additional parameter b_0 called the* bias *of the neuron.*

The *activity rule* of a single artificial neuron is the specific way we compute the output of the neuron given its inputs. Usually the activity rule has two steps. First, the response to the inputs **x** – the *activation a* of the neuron – is computed as:

$$a = \sum_{i=1}^{I} w_i x_i + b_0;$$

Second, the output *y* of the neuron is computed as a function $f(a)$ of the activation. The function $f(a)$ is also called the *activation function* while the output *y* is known as the *activity* of the neuron. In the literature there have been proposed several activation functions. Among the others, the most popular activation functions are:

1. Deterministic activation functions:

- linear:

$$f(a) = a$$

- threshold:

$$f(a) = \Theta(a) = \begin{cases} 1 & a > 0 \\ -1 & a \leq 0 \end{cases}$$

- logistic:

$$f(a) = \frac{1}{1 + e^{-\beta a}}$$

- hyperbolic:

$$f(a) = \tanh(\beta a)$$

2. Stochastic activation functions:

- Heat bath:

$$y(a) = \begin{cases} 1 & \text{with probability } \frac{1}{1+e^{-a}} \\ -1 & \text{otherwise} \end{cases}$$

- Metropolis rule. The output depends on the previous output state y, through the product $\Delta = ay$: if $\Delta < 0$, flip y to the other state, else flip y to the other state with probability $e^{-\Delta}$.

Despite it is a very simple mathematical entity, the artificial neuron has a number of nice properties that allowed for the widespread use of neural networks in so many different applications. Before describing the learning process and presenting such properties, let's show formally that we can always build a PCN whose semantic is the same of an artificial neuron. Figure 4.2 represents such a PCN.

Consider the PCN $\mathcal{AN} = \langle Lc, Tn, Cn \rangle$ defined as follows:

- Lc contains the following locations: $\{x_1, \ldots, x_I, b_0, w_1, \ldots, w_I, h_1, \ldots, h_{I+2}\}$. Where $x_1, \ldots, x_I$ are

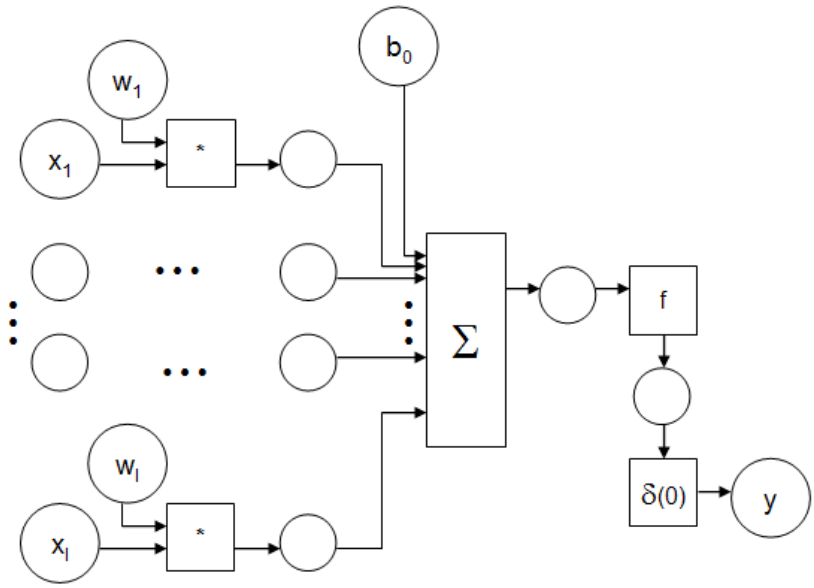


Figure 4.2: A PCN module for a generic artificial neuron.

- Tc transductions labels are: $\{+, *, delay(1), sin, G_{\mu,\sigma}\}$. As you can see in figure 2.4, the label + is used twice in the PCN. Actually, the two transductions are distinct and they must be kept separated in order to guarantee the semantical correctness of net (as we'll se later). Whenever some confusion or even a mistake can arise, it is preferable to use two distinct labels: for example $+_1$ and $+_2$.

- Cn contains all the edges between locations and transductions as depicted in figure 4.2.

[...]

Now that we are equipped with a PCN–based representation of the single artificial neuron, let's consider the learning stage. I aim at showing that we can always build a well–defined PCN that represents the learning algorithm itself.

To understand the learning of a single artificial neuron is straightforward if we introduce the concept of weight space, that is, the parameter space of the network. Given an artificial neuron with I inputs, there are at $I+1$ parameters[2]. For each selection of values of the parameter vector $\mathbf{w}$, the neural net computes a specific function, i.e. the corresponding activation function. Thus each point in the weight space corresponds to a function of $\mathbf{x}$. Now, the central idea of supervised neural networks is this. Given examples of a relationship between an input vector $\mathbf{x}$, and a target t, we hope to make the neural network *learn* a model of the relationship between $\mathbf{x}$ and t. A successfully trained network will, for any given $\mathbf{x}$, give an output y that is close (in some sense) to the target value t. Training the network involves searching in the weight space of the network for a value of $\mathbf{w}$ that produces a function that fits the provided training data well. Typically an objective function or error function is defined, as a function of $\mathbf{w}$, to measure how well the network with weights set to $\mathbf{w}$ solves the task. The objective function is a sum of terms, one for each input/target pair $\{x, t\}$, measuring how close the output $y(\mathbf{x}; \mathbf{w})$ is to the target t. The training process is simply a function minimization, and it can be carried our by adjusting $\mathbf{w}$ in such a way as to find a $\mathbf{w}_{min}$ that minimizes the objective

[2] Remember we have I weights w_i and the bias b_0.

function. Many function–minimization algorithms make use not only of the objective function, but also its gradient with respect to the parameters **w**. For instance, the backpropagation algorithm – on of the most popular in the field of ANN – efficiently evaluates the gradient of the output y with respect to the parameters **w**, and hence the gradient of the objective function with respect to **w**.

Let's describe in short how a generic learning algorithm for perceptron does work.

Formally, let's assume we have a data set of inputs $\{x^{(n)}\}_{n=1}^{N}$ with binary labels $\{t^{(n)}\}_{n=1}^{N}$, and a neuron whose output $y(\mathbf{x}; \mathbf{w})$ is bounded between 0and 1.

4.2.1 Feedforward Neural Networks

The time has come to connect multiple neurons together, making the output of one neuron be the input to another, so as to make neural networks. Neural networks can be divided into two classes on the basis of their connectivity. (a) (b) Figure 42.1. (a) A feedforward network. (b) A feedback network. Feedforward networks. In a feedforward network, all the connections are directed such that the network forms a directed acyclic graph. Feedback networks. Any network that is not a feedforward network will be called a feedback network.

The multilayer perceptron is a feedforward network. It has input neurons, hidden neurons and output neurons. The hidden neurons may be arranged in a sequence of layers. The most common multilayer perceptrons have a single hidden layer, and are known as "two–layer" networks, the number "two" counting the number of layers of neurons not including the inputs. Such a feedforward network defines a nonlinear parameterized mapping from an input **x** to an output $\mathbf{y} = y(\mathbf{x}, \mathbf{w}, A)$. The output is a continuous function of the input and of the parameters **w**; the architecture of the net, i.e., the functional form of the mapping, is denoted by A. Feedforward networks can be "trained" to perform regression and classification tasks.

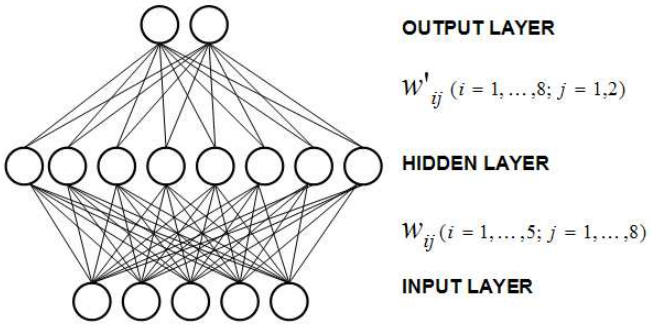


Figure 4.3: A typical two–layer feedforward neural network. There are five inputs, eight hidden units and two outputs. Network weights can be represented as two matrices...

4.2.2 Feedback Neural Networks

This section deals with neural networks that have at least one feedback connection between a pair of neurons.

Feedback networks (figure 1) can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations

The most popular class of such networks are the so–called Hopfield nets which are fully interconnected nets, i.e. each neuron has a connection with non–zero weight to any other neuron except to itself. The weights in the Hopfield network are constrained to be symmetric, i.e., the weight from neuron i to neuron j is equal to the weight from neuron j to neuron i. Hopfield networks have two applications.

The properties of a Hopfield network may be sensitive to the above choices.

The stochastic Hopfield network or Boltzmann machine (Hinton and Sejnowski, 1986) has a probabilistic activity rule.

The Boltzmann machine is time-consuming to simulate because the computation of the gradient of the log likelihood depends on taking the difference of two gradients, both found by Monte Carlo methods. So Boltzmann machines are not in widespread use. It is an area of active research to create models that embody the same capabilities using more efficient computations (Hinton et al., 1995; Dayan et al., 1995; Hinton and Ghahramani, 1997; Hinton, 2001; Hinton and Teh, 2001).

4.3 Continuous Time Recurrent Neural Networks and PCNs

In this section I describe a specific subclass of FBNNs, the so called *continuous time recurrent neural networks* (CTRNNs). I believe they deserve a different characterization because they have been playing a special role during the last few years in the Robotic research field.

Many authors prefer to describe the *activity* of each neuron of the net in terms of differential equations (see equation 4.1 below) instead of in terms of the messages passed through a graph of neurons[3]. Such a more strict mathematical representation allow us to build the corresponding PCN, which can be used directly whenever the neural system is described as a dynamical system evolving over time. The goal of this section, thus, is to take advantage of the peculiar properties of CTRNNs and provide a compact (and, hopefully, more efficient) representation for them within the PCNs framework.

Informally, a CTRNN is a neural system that comprises N different, fully interconnected[4]

[3]This choice is not only a matter of preference, of course. In fact there are some properties of the neural system that can be more conveniently expressed if we look at it as a dynamical system. Sometimes our goal is to exploit such properties; and in these cases we are forced to adopt this approach.

[4]Feedback loops are also allowed.

artificial neurons $\{q_i\}_{i\in 1,\ldots,N}$. Each neuron is further connected to some neuron–like elements called *input of the net*. Formally, a CTRNN $\mathcal{R}$ is defined as the quadruple $\langle Q, I, W, W'\rangle$, where:

- $Q = \{q_i\}_{i=1}^{N}$ is the set of the nodes of the net;
- $I = \{I_k\}_{k=1}^{S}$ is the set of the inputs of the net;
- $W : (q_i, q_j) \in Q \times Q \rightarrow w_{ij} \in \mathbb{R}$ is a function that defines the weights associated to the connection between each pair q_i, q_j of nodes;
- $W : (q_i, I_k) \in Q \times I \rightarrow w'_{ik} \in \mathbb{R}$ is a function that defines the weights of the input I_k over the node q_i.

The state of each neuron q_i at time t is described by a function $y_i(t)$ called *activation* of the neuron[5]. The reader should note many similarities between the terminology proposed here and the one introduced above for artificial neuron in general; this should have been easily expected.

The semantics of the nets – i.e. its dynamical behavior – is the solution of a system of differential equation whose i-th equation is:

$$\dot{y}_i = f_i(y_1, \ldots, y_N) = \frac{1}{\tau_i}\left(-y_i + \sum_{j=1}^{N} w_{ji}\sigma(y_j - \theta_j) + \sum_{k=1}^{S} w'_{ki} I_k\right), \tag{4.1}$$

where σ is the logistic sigmoidal function introduced above, τ_i is a time constant[6], and θ_i is a bias term associated to each neuron.

If we constraint the matrix w_{ij} to to be symmetric (with zero diagonal elements) we can use a well known result of Hopfield (1984) which states that such networks could be used as associative memories, with each pattern stored as a different equilibrium point attractor of the

[5]The dependence on t is usually omitted if no ambiguity can arise.

[6]If the equation is considered a model of the biological neuron, the time constant is related to some properties of the membrane of the neuron. In a pure mathematical framework it provide a useful rescaling factor to each equation.

network. This is an “attractive” property for robotic researchers since it could be used to link specific configuration of the environment to corresponding desired response. In fact, as we’ll see in chapter **??**, it is possible to use equations 4.1 as motor behavior controller.

In this section we are interested in studying equations 4.1 in the general framework of the methods for *ordinary differential equations*. As stated in chapter 2 the formal syntax of PCNs was defined with the constraint that PCNs should be able at least to express (stochastic) dynamical systems – i.e. systems of (stochastic) differential equation – and thus we are sure there exist a suitable PCN for any system like 4.1.

Actually we are not satisfied of this guarantee and want something further: we want a *methodological* example of how to build the PCN. Of course we can reason about the general case with N neurons and I inputs, but the considerations still hold if we reason about a special case with, e.g., 2 interconnected neurons with one input I. All the formulas and pictures will be definitely clearer and this section will be more useful for practical uses. Let’s consider the system whose behavior is:

$$\begin{cases} \tau_1 \dot{y}_1 = -y_1 + w_{11}\sigma(y_1 - \theta_1) + w_{21}\sigma(y_2 - \theta_2) + w'_1 I \\ \tau_2 \dot{y}_2 = -y_2 + w_{12}\sigma(y_1 - \theta_1) + w_{22}\sigma(y_2 - \theta_2) + w'_2 I \end{cases} \tag{4.2}$$

In order to implement on a digital computer the equations above, we must use a numerical integration method. Let’s adopt the same method we used in the example 2.3; and we obtain:

$$\begin{cases} y_1^{n+1} = y_1^n + \frac{\Delta t}{\tau_1}\left(-y_1^n + w_{11}\sigma(y_1^n - \theta_1) + w_{21}\sigma(y_2^n - \theta_2) + w'_1 I^{n+1}\right) \\ y_2^{n+1} = y_2^n + \frac{\Delta t}{\tau_2}\left(-y_2^n + w_{12}\sigma(y_1^n - \theta_1) + w_{22}\sigma(y_2^n - \theta_2) + w'_2 I^{n+1}\right) \end{cases} \tag{4.3}$$

where n represents the step of the computation and Δt the so called step size. Interestingly enough, a generalization of equations 4.3 can be adopted as the fundamental equation of another class of recurrent neural networks, namely the *discrete time recurrent neural networks*

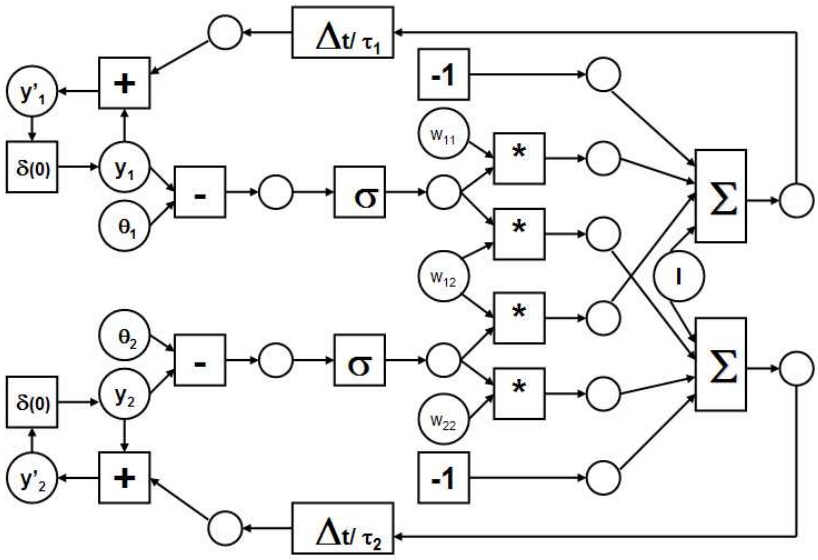


Figure 4.4: PCN that represents an implementation of a CTRNN

DTRNN[7].

It is quite straightforward to build the PCN that represent equations 4.3; it is represente in figure 4.4.

All the transduction are basic transliterations, except for two unit delays $\delta_1(0)$ and $\delta_2(0)$ which are necessary to avoid algebraic loops. The only location that have a physical meaning are $y_1, y_2, \theta_1, \theta_2, I_1, I_2$ and all the weights. Their domains are usually some interval of $I \subseteq \mathbb{R}$. The inputs of the PCN are the two inputs of the net while we are interested (as output) to the traces associated with y_1 and y_2.

[7]This (supposed) equivalence between DTRNN and any algorithmic implementation of CTRNN is an interesting topic but it would lead far beyond the scope of this thesis.

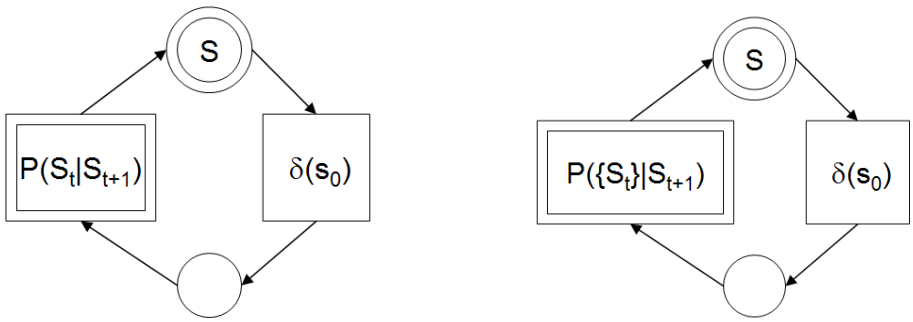
4.4 Markov Models and PCNs

In this section I briefly recall a number of results presented already in St–Aubin's thesis. They show that any Markov model can be represented by a suitable PCN, and this is valuable[8] because in practice many robotic systems naturally make use of the Markovian hypotheses about the probabilistic dependance between any two consecutive states of the system.

There exist four distinct cases of Markov models: they are obtained by combination of both discrete vs. continuous state space and discrete vs. continuous time. Let's start from the easiest one; if both time and state space are discrete, then the model $\mathcal{MC}$ is called *discrete time Markov Chain* (DTMC). A DTMC is a tuple $\langle S, s_0.\mathcal{P}\rangle$, where S, s_0 and $\mathcal{P}$ represents the finite set of states ($|S| = n$), the initial state, and the probability transition respectively. It is quite straightforward to build the corresponding PCN module $\mathcal{MC}_{PCN}$ of a given $\mathcal{MC}$; it is simply $\langle\{S\},\{\delta(s_0),\mathcal{P}_{PCN}\},C_{PCN}\rangle$. This means that the set of locations of $\mathcal{M}_{PCN}$ contains one location S whose domain is the set of all possible states of the DTMC: $\{1,2,\ldots,n\}$, and each value of the location encodes for the corresponding state of the DTMC. $\mathcal{MC}_{PCN}$ contains only one deterministic transduction – the unit delay $\delta(s_0)$, and one generator $\mathcal{P}_{PCN}$ following the probability distribution $\mathcal{P}$. We need $\delta(s_0)$ not only in order to avoid an algebraic loop but also to model the Markovian property of the Markov chain; in fact, the unit delay guarantees that the state S_t – i.e. the value of the location S in the domain at time t – depends only on state S_{t-1}. The generator in $\mathcal{P}_{PCN}$ is equivalent to the probability transition matrix of $\mathcal{MC}$; thus, given the current S_t, the generator provides the probability distribution of the next possible S_{t+1}. C_{PCN} contains only three connections that are represented in figure 4.5(a).

If we let the state space to be continuous while keeping the time discrete, i.e. we consider the so called *discrete–time Markov Processes* (DTMP), then it is still very simple to build the corresponding Markov–process like PCN $\mathcal{MP}_{PCN}$. The only difference with respect to the

[8] These valuable results are quite relevant to the idea discussed in this chapter, thus I believe it is worthwhile to recall them here.



(a) PCN equivalent of a DTMC (b) PCN equivalent of a DTMP

Figure 4.5: Equivalence between PCNs and discrete–time Markov models

previous case is that instead of a probability transition matrix, we must introduced a probability measure $P(\{S_t\}|S_{t-1})$ over sets of states. The representation of such a DTMP is shown in figure 4.5(b). The location S has a continuous domain (e.g. $\mathbb{R}$) rather than a discrete one. Moreover the generator is now defined on a set of states $\{S\}$.

The case of continuous–time and discrete state space is called *continuous–time Markov Chain* (CTMC). In a CTMC, the state transitions may occur at any time, with a given probability rate. In order to manage this transition rate, it is usual to define the so called *rate matrix* $R_{ij}^t(s)$ that represents the transition probability from state i to state j and from time t to time $t + s$. Often, the transition probabilities are independent from the initial time t – the chain is called time homogeneous – and thus $R_{ij}(s)$ denotes the transition probability from i to j over s time period. In the most common type of time homogeneous CTMCs the time between transitions is exponentially distributed. Since the exponential distribution is memoryless, the future outcome of the process depends only on the present state and does not depend on when the last transition occurred or what any of the previous states were, and this allows the Markov property to still hold. In such cases, the rate matrix is actually a three–dimensional $R_{ij}(s)$ where, for each pair of states i and j, there is the corresponding rate parameter of the

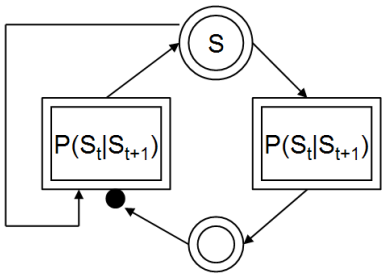


Figure 4.6: PCN equivalent of a Continuous Time Markov Chain

exponential distribution.

Since our goal is to build the PCN corresponding to the chain, let's start from a CTMC $\mathcal{CMC} = \langle S, s_0, R \rangle$, whit $|S| = n$. The equivalent $\mathcal{CMC}_{PCN}$ is $\langle \{S\}, \{Rate, P\}, C \rangle$, where S is the only location of the system and its domain $\{s_0, s_1, \ldots, s_n\}$ encodes the n states of the system. *Rate* is a stochastic event generator following an exponential distribution with state–dependent rate $R(t)$ that triggers an event when the transition condition has been completed; P is the generator following the distribution $P(i, j) = R_{i,j}(s)$ for all s in domain(S) which causes the system to transition probabilistically to a new state s0 when: 1) the system is in state s, and 2) an event signifying the completion of the race condition has occurred. This general situation, which applies to any CTMC with discrete state space, is represented in Figure 4.4.

The final, last case is the pure analog continuous time Markov processes (CTMP) which is a special case of stochastic differential equation. The equivalence of CTMPs and PCNs was showd via the equivalence of stochastic integration

4.5 Planning, Markov Decision Processes, Reinforcement Learning

Chapter 9 of St-Aubin (2005) discusses the problem of control synthesis and its relationships with PCNs framework. The goal is to introduce the reader to specific techniques that are especially well suited to synthesize controllers when in the presence of a PCN model. The focus of that chapter is on *(Partially Observable) Markov Decision Processes*, or (PO)MDPs, which are quite popular methods within the AI community because they are effective algorithms to compute optimal policies. St–Aubin showed the existence of a subclass of PCN models, which he called *synchfin*–PCN, and which has a one–to–one correspondence to the class of all MDPs. This result is valuable from the point of view of this thesis because it

As mentioned earlier, policies can be viewed as controllers; hence computing a policy can be seen as control synthesis. It would be extremely valuable to be able to merge the modeling simplicity and power of the PCN framework with the control synthesis capabilities of MDP.

Maybe one of the most common problems in mobile robotics is *planning under uncertainty*, which is also known as *decision–theoretic planning* (DTP). In short, a DTP approach is useful in those systems whose dynamics can be modelled as stochastic processes and where an agent, acting as a decision maker, can influence the system's behavior by performing (uncertain) actions. Resulting from the Markov property, the current state of the system and the choice of the action by the agent jointly determine a probability distribution over the possible next states. It is usually assumed that systems evolve in stages, where actions cause

Reinforcement learning is learning what to do–how to map situations to actions–so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent

rewards. These two characteristics–trial-and-error search and delayed reward–are the two most important distinguishing features of reinforcement learning.

Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem. Any method that is well suited to solving that problem, we consider to be a reinforcement learning method. The basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting with its environment to achieve a goal. Clearly, such an agent must be able to sense the state of the environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment. The formulation is intended to include just these three aspects–sensation, action, and goal–in their simplest possible forms without trivializing any of them.

We do not intend to cover all those areas here, but rather we wish to introduce the reader to specific techniques that are especially well suited to synthesize controllers when in the presence of a PCN model.

A Markov decision process $\mathcal{MDP}$ is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, P, R \rangle$, where $\mathcal{S}$ is a finite set of states of the system, and where states are defined as a description of the system at any point in time. In a MDP, these states can be exactly identified by the agent, i.e., at any given time the agent knows exactly which state it is in. $\mathcal{A}$ is a finite set of actions from which the agent can choose; P is the state transition model of the system which is a function mapping from elements of $\mathcal{S} \times \mathcal{A}$ into discrete probability distributions over $\mathcal{S}$; and R is a stationary reward function mapping from $\mathcal{S} \times \mathcal{A}$ to $\mathbb{R}$. $R(s,a)$ specifies the immediate reward gained by the agent for taking action a in state s. Actions induce stochastic transitions, with $P(s,a,t)$ denoting the probability with which state t is reached when, at the previous time step, action a is performed at state s. It is to be noted that the transitions of the model specify the resulting next state using only the state and action at the previous time step. This therefore assumes that the next state is solely determined by the current state and the current action and corresponds

to the Markov assumption discussed earlier. It is worth mentioning that not all systems are Markovian in nature. The Markov assumption is merely a property of a particular model of that system, not of the system itself. However, one should note that the Markovian assumption is not too restrictive, since any non-Markovian model of a system can be converted to an equivalent Markov model. In the field of control theory, this conversion is referred to as the conversion to state form [Lue79]. A *stationary policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$ describes a particular, time independent, course of action to be adopted by an agent, with $\pi(s)$ denoting the action to be taken in state s. It is often assumed that the agent acts indefinitely (an infinite horizon) but the finite horizon case has also been studied extensively. In the finite-horizon case however, the optimal policy is typically non-stationary: the agent's choice of action on the last step of his life will generally be very different than when it has a long life ahead of it.

We will, in this short presentation of the MDP framework, assume infinite horizon, unless explicitly stated. A possible way to assess the quality of different policies is to adopt an expected total discounted reward as the optimality criterion wherein future rewards are discounted at a rate $0 \leq \beta < 1$, and the value of a policy is given by the expected total discounted reward accrued. The expected value $V_\pi(s)$ of a policy π at a given state s satisfies [Put94]:

$$V_\pi(s) = R(s, \pi(s)) + \beta \sum_{t \in \mathcal{S}} P(s, \pi(s), t) V_\pi(t) \tag{4.4}$$

4.6 Kalman Filter and Bayesian Filtering as PCNs

The Kalman filter (KF) is a very powerful mathematical tool that is playing an increasingly important role in mobile robotics, for example as *adaptive filtering device* for localization[9]. Actually, KF is not the cutting edge of stochastic estimation since it has been around for about 40 years (Kalman 1960). However, it turned out to be an *optimal* estimator for

[9]Of course, this is not the only application of KF to robotics, yet this is the most popular in the last few years. However the present discussion does not rely on one this specific example but rather on the mathematical properties of the KF as filtering device in general.

a large class of problems and a very effective and useful estimator for an even larger class; moreover, it is extremely easy to implement and pretty fast in many practical applications. All this resulted in a widespread use of the filter and explains its popularity. The following constructive demonstration of the equivalence between PCNs and KF, then, could result in a useful contribution.

The following presentation – originally due to Sorenson (1970) – is the common way to introduce and explain the Kalman filter and it is very helpful to catch on to the basics of the topic. For more extensive discussion on KF and stochastic estimation in general the reader is referred – for example – to (Lewis 1986) and (Kailath et al. 2000); a very helpful tutorial on KF was written by Welch and Bishop (2001).

Chapter 5

PCNJ: A Visual Programming Environment for Probabilistic Constraint Nets

In the previous three chapters I introduced the PCN framework and I described a model–checking approach to behavioral verification for Robotics. Furthermore, several examples have been discussed in order to make it clear how many interesting features and tools PCNs provide for robotic researchers. However, everything would remain in the realm of abstract discussions if we didn't provide an effective toolbox that people can rely on when designing and modelling their robotic systems.

This chapter describes an integrated programming environment called PCNJ – that stands for *Probabilistic Constraint Nets in Java* – which supports probabilistic constraint net modelling, simulation, and animation for any kind of hybrid systems. My contribution to PCNJ was twofold; as a Visiting Scholar at the Laboratory for Computational Intelligence[1] I collaborated with Alan Mackworth and Lee Leif Chang on the designing and development of the fundamental packages of PCNJ[2]. Moreover, during the last few months, I've been involved in testing the pre–release version of PCNJ. Several tools have been added to PCNJ as side effects of the implementation of many of the examples described in this thesis.

[1] At the University of British Columbia, Vancouver B.C., CA

[2] I focused mainly on the two packages `core` and `simulation`.

Some insightful examples based on “concrete” robotic problems are presented in the next chapters and for each of them a PCN–based program was created in PCNJ. The experiments conducted on them confirm the effectiveness of PCNJ as a tool for hybrid system modelling and real-time simulation.

5.1 Concepts of PCNJ

PCNJ is thought to be an *integrated development environment* (IDE) for people that want both to build a PCN and to simulate its dynamical evolution in order to verify the behavior of a (stochastic) hybrid system. PCNJ allows for the modelling of PCNs that either are pure software simulations or are coupled with some physical device – for example a software module that is connected to (and controls) a real mobile robot[3].

IDEs are popular and useful pieces of software that assists computer programmers to develop other software. IDEs normally comprise a source code editor, a compiler or an interpreter, and (usually) a debugger. Moreover, numerous tools are provided to further simplify the “construction” of new software. IDEs are becoming an indispensable support for developing large pieces of software composed of many independent parts. Typically, IDEs are not general–purpose environments since each IDE is devoted to a specific programming language, even if there exist a few multiple–language IDEs[4] such as the Eclipse IDE, NetBeans or Microsoft Visual Studio. Thus, despite the availability of many professional tools, they are not suitable for modelling PCNs directly; we cannot rely on them and so we definitely must face the problem of building a specific IDE for our purposes.

Since our goal is to build an IDE for PCNs creation and simulation, then the first step is to specify *which is* the *programming language*. So far, in fact, we have not introduced formally

[3] In such cases the PCN module is the controller of a real robotic body that senses and interacts with its environment.

[4] Usually these are professional IDEs and are tailored for the most popular procedural and/or object–oriented programming languages such as C/C++/C#, Java, Visual Basic

any programming language. As described in chapter 2, PCN framework deals with systems of equations defined on a dynamics structure $\mathcal{D}(\mathcal{T},\mathcal{A})$ where $\mathcal{T}$ and $\mathcal{A}$ denote an abstract time structure and an abstract domain structure, respectively. Even if it is quite intuitive how it is possible to move from such an equation–based specification language to a concrete formal programming language, I believe it is worth describing the transition in a clear way in order to avoid unwarranted pitfalls. The nice graphical representation for PCNs introduced in section 2.2.2 clues us in that our language may be a visual programming language[5] (VPL).

Thus, let's start from the graphical representation of a PCN as a bipartite graph. The graph that represents a specific PCN includes a group of nodes (the locations) that store the value of variables over time. Some other nodes (the transductions) represent functional relationships among variables. Some transductions – the event generators – play a special role since their output is an event that can trigger other transductions. Event generators whose outputs are produced at a fixed time rate, can be used as clocks[6] and can be attached to both primitive transductions and generators or to other modules.

The above graph–based picture of PCN closely resembles that of a dataflow computing model (DFCM). Unlike the more standard, control–flow computing model (CFCM), DFCM is based on the flow of information between data processing entities, instead of the flow of control between instructions; more specifically, DFCM assumes that a program is a data–dependency graph whose nodes denote operations and whose edges denote dependencies between operations. DFCM executes any operation denoted by a node as soon as its incoming edges have the necessary operands (see Jagannathan (1996) for a thoroughly description of dataflow computing approaches). This similarity between PCNs and dataflow graphs was the *bridge* between dynamical systems and programming languages we had been searching for.

Given the above intuitive idea of *how* the "PCN programming language" should look like,

[5] See Chang (1990) and Burnett (1999) for an introduction to the most important concepts of visual programming language.

[6] Note that we are not making any assumption about parallelization and synchronization of computation. There can be either just one or more clocks and they can be either dependent or independent to each other.

let's point out some critical issues we'd better consider carefully in order to avoid conceptual misleading. The original syntax of PCNs deals with abstract mathematical entities; for example transductions map input trace spaces to output trace spaces. Hence, the fundamental concept of time is *hard–coded* into the abstract concept of dynamics structure. Any software implementation of this abstraction must address the problem of "unveiling the time" and make it explicit what can trigger the computation in the actual programs. Mathematically, both primitive and compound transductions can not alter the time structure underlying a trace whereas event generators can do it – in fact they are used to link continuous and discrete systems together. Thus, a plausible solution would be to associate a *thread* of computation to each syntactical counterpart (in the programming language) of the the event generators, and let them trigger a (non empty) subset of transductions[7] We'll call these elements *clocks*, to which we associate a fixed firing–rate. We require at least one clock to be specified for each well–defined program; this is somehow equivalent to the abstract requirement that dynamics structures relies on at least one abstract time structure.

A final issue we must be aware of is possibility of define independent PCN modules or sub–nets; the semantics of the modules should be preserved during the computation.

5.1.1 The $\mathcal{L}_{\mathcal{PCN}}$ Visual Language

In this section I introduce a visual programming language called $\mathcal{L}_{\mathcal{PCN}}$ – based on the PCNs framework – which underlies the PCNJ IDE. A well–defined $\mathcal{L}_{\mathcal{PCN}}$ program provides a software implementation of the corresponding abstract PCN, i.e. the ordered set of the values of a specific $\mathcal{L}_{\mathcal{PCN}}$ variable over time is a sample[8] of the trace associated to the corresponding location.

Let's now define the basic syntactical elements of the language that we called $\mathcal{L}_{\mathcal{PCN}}$:

[7] There must be at least on transduction per thread, i.e. two distinct thread cannot fire on the same transduction.

[8] If the PCN is defined on a continuous time structure then this set is actually a sample. It the time structure is discrete the this set coincides with the trace.

Circles : these are both the *constants* and the *variables* of $\mathcal{L}_{\mathcal{PCN}}$. A circle denotes one specific *location* of the PCN, and it stores the *value* of the location over time. The domain of the circles is the domain of the corresponding location. The *sort* a circle is the corresponding type of domain. We call *constants* those circles associated to an *input location* and *variables* all the others. Those circles for which there exists an arrow from a double border square have themselves a double border, i.e. they are the stochastic locations.

Squares : these are *built–in* operators that can act on the values of input circles and whose output updates the values of the output circle. A circle is an *input circle* of the square if there exists an arrow from the circle to the square. A circle is an *output circle* of the square if there exists an arrow from the square to the circle. Squares corresponds to the basic *transductions* of the PCN. We adopt the convention that *generators* have a double border while *transliterations* have a single border. Each square have a specific signature and it is possible to draw arrows between a circle and a square only if the type of the circle satisfy the signature of the square.

Clocks : these are a special type of squares. They corresponds to the *event generators* of the corresponding PCN. Each clock has a specific *firing–rate*. Arrows can be drawn from a clock to a transduction directly.

Arrows : these can connect any circle to a square and viceversa. It is not possible to connect neither a circle to another circle nor a square to another square. It is possible to connect a clock to a square. Arrows correspond to the connections of the corresponding PCN. The set of the arrows of a $\mathcal{L}_{\mathcal{PCN}}$ program must satisfy the constraints imposed on the connections of the corresponding PCN.

To summarize, the above syntactical elements can be combined together in order to define a well–formed $\mathcal{L}_{\mathcal{PCN}}$ program:

Definition 5.1 ($\mathcal{L}_{\mathcal{PCN}}$ program) *Let's consider the bipartite graphs $\mathcal{G}$ whose vertices are either circles or squares (or clocks), and whose edges are the arrows. We say that $\mathcal{G}$ is a well–defined $\mathcal{L}_{\mathcal{PCN}}$ program iff the corresponding PCN is a well–defined Probabilistic Constraint Nets. Furthermore, we require that* (1) *$\mathcal{G}$ must contain at least on clock, and* (2) *there must exist exactly one arrow pointing to a each transduction and starting from one of the clocks in $\mathcal{G}$.*

Definition 5.2 (PCN programming language ($\mathcal{L}_{\mathcal{PCN}}$)) *The PCN programming language is the set of all the well–defined $\mathcal{L}_{\mathcal{PCN}}$ programs.*

The execution of a PCN program is a specific computation given a set of sequences of input values and a (possibly infinite) sequence of *firing* signals from each clock to the corresponding transductions. It is easy to show that, given the previous definitions, the semantics of the program is exactly that of the constraint net, given that we guarantee a correct initialization of the computation. If all the values of the input locations are properly defined at the initial step then, at each subsequent step, some of the undefined locations are computed and eventually, the constraint net outputs are computed.

Now that we are equipped with the programming language $\mathcal{L}_{\mathcal{PCN}}$ we can face the problems of designing the *compiler/interpreter* for it. Because we defined a graph–based approach that is very similar to the dataflow computing model, we adopted the approach of interpreting by means of a simulation of the evolution of the system. In the section 5.4 I discuss thoroughly how PCNJ simulate any PCN, that is to say how the *interpreter* does work.

In PCNJ, the computation is triggered by a PCN module component is often both connected with and driven by a clock. To evaluate the module correctly, the transductions contained in the module have to be triggered in proper sequence. CNJ uses the transduction scheduling algorithm to figure out a right dependency relationship within the transductions. Then the clock triggers those transductions one by one in that order. This approach allows the

computation of constraint net modules to work as demand-driven dataflow, and it works well.

5.1.2 PCNJ and its Relationships with other Visual Programming Languages

Before presenting the details of PCNJ, I propose in this section a very brief overview of the most common visual programming languages to which $\mathcal{L}_{\mathcal{PCN}}$ and PCNJ should be compared:

Matlab/Simulink Matlab/Simulink is a visual programming and simulation environment for continuous and discrete control systems. It enables users to build graphical block diagrams, simulate dynamic systems, evaluate system performance, and refine their designs. It is currently the most popular tool for control system modeling and simulation. However, it is not suited for hybrid system modeling in constraint nets. There are three reasons for this:

1. First, Simulink is unable to support an event–based time structure, which is an important characteristic of hybrid systems.
2. Second, although it supports bottom-up modeling well (by grouping), it does not support top-down and middle-out modeling methods, which are helpful for some users.
3. Third, in Simulink, all the system models are stored in MDL format (Model Description Language). In addition, since CN has a different graphical representation from Simulink's models, the MDL file format is not able to store constraint net models.

LaBVIEW : LabVIEW is a platform and development environment for a visual programming language named "G". LabVIEW is commonly used for data acquisition, instrument control, and industrial automation The programming language "G", is a dataflow language. Execution is determined by the structure of a graphical block diagram (the

LV–source code) on which the programmer connects different function–nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, “G” is inherently capable of parallel execution. Multi-processing and multi–threading hardware is automatically exploited by the built–in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

One main benefit of LabVIEW over other development environments is the extensive support for accessing instrumentation hardware. Drivers and abstraction layers for many different types of instruments and buses are included or are available for inclusion. These present themselves as graphical nodes. The abstraction layers offer standard software interfaces to communicate with hardware devices. The provided driver interfaces save program development time.

Unfortunately, LabVIEW is a proprietary product of National Instruments. Hence, LabVIEW is not managed or specified by a third party standards committee such as the ANSI for C. Obtaining a fully compatible and up to date LabVIEW platform requires purchasing the product. Thus this very promising approach is not suitable for the purposes of the present thesis.

5.2 System Requirements for PCNJ

Because PCNJ is thought to be an IDE for a visual programming language, it should be able at least to support users to *draw* a program by means of PCN graphical primitives.

We impose the following requirements on our modeling and simulation environment:

1. it should enable developers to interactively pick components, and place them onto a work area. These components are CN’s atomic nodes: locations, transductions, connections, and modules;

2. connecting has to be accomplished in a way where events and data can be exchanged correctly among the components;

3. the convenient interactive customization of bean properties should be supported by the environment;

4. there has to be a method to check each CN node's dynamic values. For instance, users might wish to see a location's changing values while a simulation is running.

5. such a modeling and simulation environment has to be simple to use and execute. Also the designed model should be reusable as a new component in any other hybrid system. In this case, a very complicated system can be built by assembling some less complicated components.

In such a visual programming environment, users "draw" constraint net programs, instead of writing code for them. The look and feel is intended to resemble the style of some popular drawing tools such as Adobe Illustrator, MS Painter, and Unix xfig to support constraint net designing. In addition, to make the GUI respond as quickly as possible, we adopt multi-threaded programming to minimize the response time to users' action. In Constraint Nets, a model possibly consists of dozens of modules that are hierarchically located in different levels. To support as many modules as possible, CNJ uses Multiple Document Interface (MDI) to display each module in a child window. Thus, every module component corresponds to a child window in the MDI desktop.

5.3 Software Architecture of PCNJ

In this section I describe the software architecture of PCNJ. Of course, many details will be omitted and the focus will be on

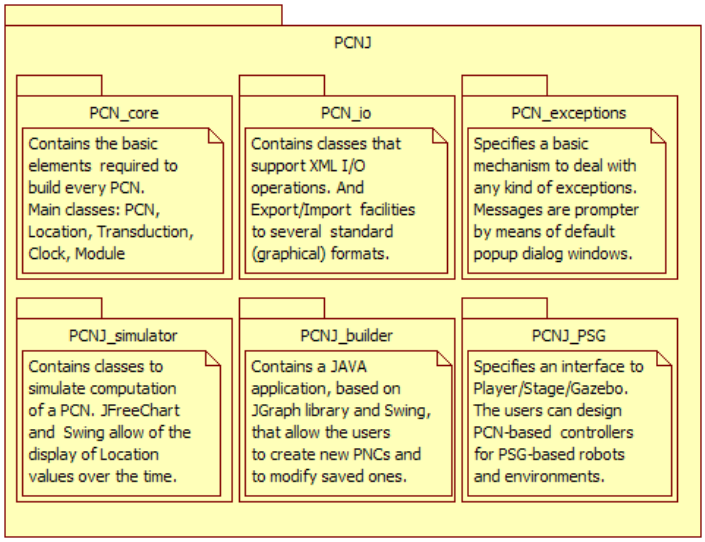


Figure 5.1: Overview of the packages of PCNJ

The approach we adopted to develop PNCJ is to use as much as possible already developed open–source (or freely available) software. In particular we

An overview of software architecture of PCNJ is in figure 5.3. Currently, PCNJ comprises 6 packages, the most important of which is the `core` package. It contains classes that are in ono–to–one correspondence with the syntactical elements of PCNs framework. All the other packages depends on `core` and provides the following features to PCNJ:

`pcn.io` : contains classes that support I/O capabilities for $\mathcal{L}_{\mathcal{PCN}}$ programs. It is possible to save and open each graph associated to a $\mathcal{L}_{\mathcal{PCN}}$ program. We adapted the XML–based language introduced by Song (2002) and thus it is possible to save hierarchical description of PCNs into textual files by means of abstract XML syntax. Moreover it is possible to import/export $\mathcal{L}_{\mathcal{PCN}}$ programs as pure JAVA objects because every class in the `pcn.core` package implements the *Serializable* interface[9]. A further I/O capability is provided by the `pcn.builder` package and thus it is possible to save $\mathcal{L}_{\mathcal{PCN}}$ programs

[9] See some JAVA reference for further detail.

as image.

pcn.exceptions : contains several utility classes to handle almost any of exception that can be generated within PCNJ.

pcn.simulator : contains classes the allow us to generate execution traces for each $\mathcal{L}_{\mathcal{PCN}}$ program. Classes in package exploit the properties of *Clocks* to be autonomous threads of computation; they start the computation and *listen* to the changes produces to the value of output traces. JFreeChart and Swing allow for the displaying of traces over time.

pcn.builder : contains classes that allowed us to create *PCNJ–Builder* that is, actually, the graphical user interface of the IDE discussed in this chapter. Most of functionalities of *PCNJ–Builder* are provided by the free library *JGraph* on which we strongly rely. JGraph provides a range of graph drawing functionality for either client–side or server–side applications. JGraph has a powerful API that allows for the visualization, manipulation, automatic layout managing and, finally, it provides tools to make some analysis of graphs. JGraph complies with all of Swings standards, such as *pluggable* look and feel, data transfer, accessibility, internationalization and serialization. Furthermore, advanced features such as undo/redo, printing and XML support, the standard Swing designs are also included. JGraph also complies with the Java conventions for method and variable naming, source code layout and javadocs comments.

pcn.psg : contains classes useful to simulate a PCN–based controller that govern either a real robot or a simulated body/environment. This package should be considered an API to the open–source, popular robotic interface PLAYER/STAGE/GAZEBO.

5.3.1 PCNJ–core package

The most important part of PCNJ is, definitely, the `core` package. It contains one specific class for each syntactical element of $\mathcal{L}_{\mathcal{PCN}}$ hence it can be used in two different ways:

- people that want to design PCNs by means of the *PCNJ–Builder* user interface can simply drag and drop graphical tokens and draw arrows between them. Hence, each class of the `core` package might be inherited by a corresponding class in the `builder` package that have some further display/visualization capabilities.
- people that

5.4 Simulations of PCNs within PCNJ

One of the most important feature of the current version of PCNJ is the simulation of the dynamical evolution of any $\mathcal{L}_{\mathcal{PCN}}$ program that stands for an abstract PCN.

The simultaneous evolution of several traces – each with its own time structure – is deeply founded upon the idea of parallelism. Transliterations encodes functional mappings between trace spaces and are used to constraint the evolution of the output trace space based on the actual state of the input locations. If we really want to build a software simulation of these difficult abstract concepts we can not avoid to use parallel computation. Hence we developed a multi–thread mechanism within PCNJ in order to – let's say – *execute* a $\mathcal{L}_{\mathcal{PCN}}$ program. The basic idea is well schematized in figure 5.4.

Each Clock is an independent thread of computation with its own *firing–rate*; thus the life–cycle of a that specific thread is as follows:

1. fires an event for each transduction that are connected with an arrow exiting from the clock;

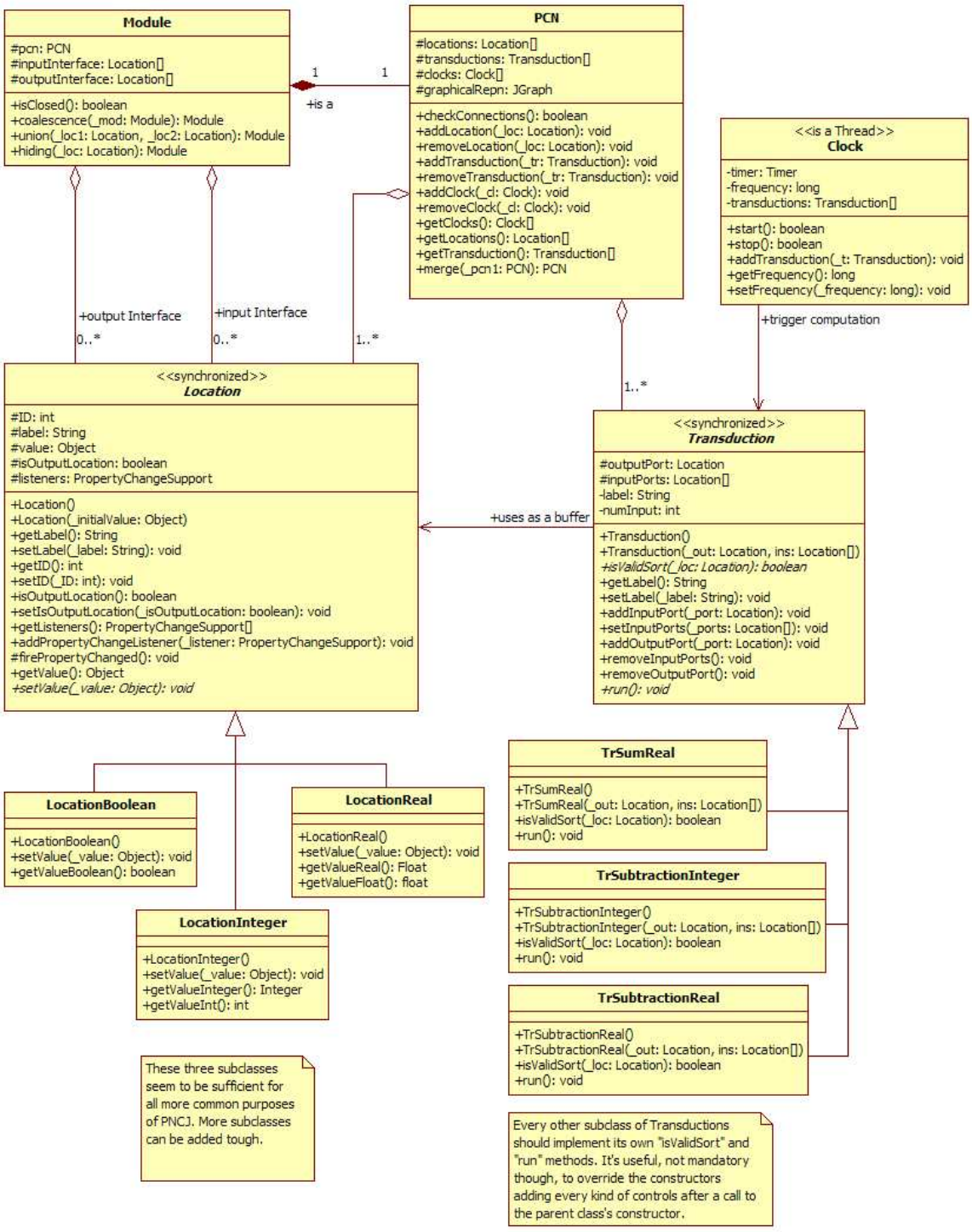


Figure 5.2: UML diagram of the `core` package in PCNJ

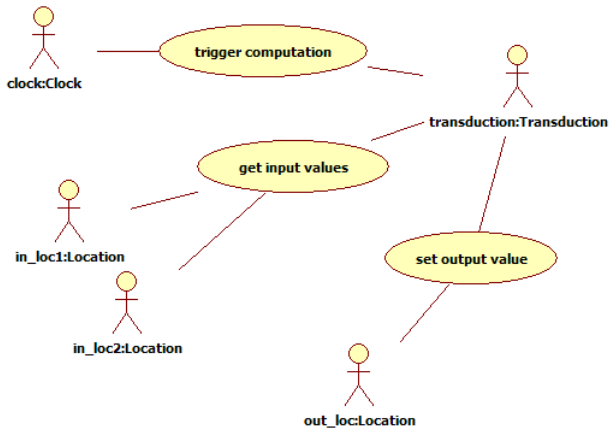


Figure 5.3: A schematic representation of the mechanism for activate the computation of a Transduction in PCNJ

2. tells the java virtual machine that it can sleep for a Δt interval of time (depending on its *firing–rate*;

3. wait until the java virtual machine re-activate the thread, and start again from point 1.

When fired by a Clock, then the transduction should simply read the inputs from its input locations and compute its output which will be set as the new value of the output location. The scheduling mechanism for the activation of the Clocks is managed directly java virtual machine. Proper *locks* are defined over the variables in order to guarantee the correctness of the computation.

The final problem we must face is the scheduling of the activatio sequence of the transductions controlled by a single clock. In PCNJ we adapted an algorithm proposed by Song (2002) which is described as follows.

This algorithm aims at specifying a correct order to drive the transductions; it is based on the *topological sorting algorithm*. Topological sorting is a natural problem in many algo-

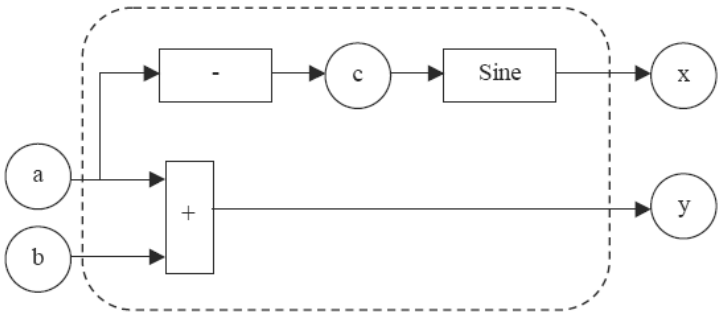


Figure 5.4: (Taken from Song (2002)) An example of a PCN whose transduction are activated by a single clock; and thus they must be scheduled correctly in order to simulate the PCN correctly.

rithms on directed acyclic graphs (DAG). Topological sorting orders the vertices and edges of a DAG in a simple and consistent way. It can be used to schedule tasks under dependency constraints. The problem of topological sorting is described as follows:

Input : A directed, acyclic graph $G = (V, E)$ (also known as a partial order).

Problem : Find a linear ordering of the vertices of V such that for each edge (i, j) in E, vertex i is to the left of vertex j.

The topological sorting problem is also applicable to constraint net graphs. Suppose we have a set of transductions to be driven in a PCN module, and certain transductions must be computed before other transductions. These dependency constraints thus form a constraint net (also a directed graph). The transduction scheduling algorithm searches for an order to execute the transductions, such that each is performed only after all of its previous transductions are executed. In the implementation, it utilizes the breadth-first algorithm to transverse the constraint net graph, but in a backward way (from output interface locations to input interface locations). The algorithm picks vertices in hierarchical levels with the output interface locations as roots. That is, if a vertex has an out–degree–count 0 it can be next in the topological order. Then, the algorithm removes this vertex and looks for another vertex with an

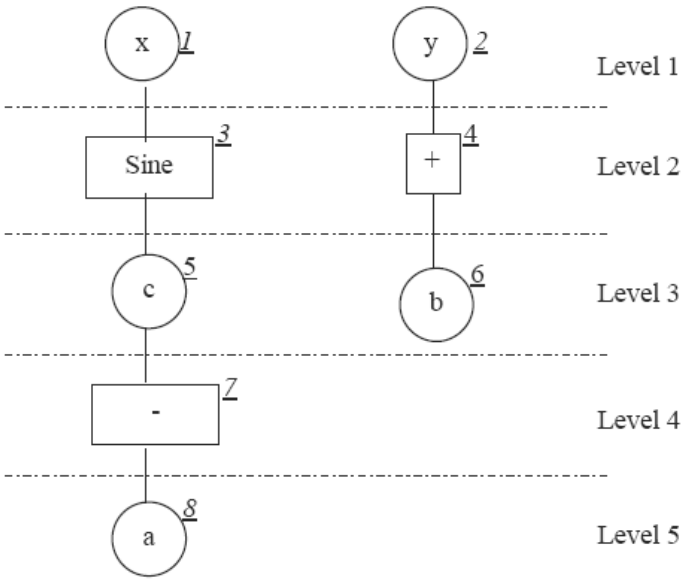


Figure 5.5: (Taken from Song (2002)) The hierarchical tree after sorting.

out–degree–count 0 in the resulting DAG. It repeats this until all vertices are added to the topological order. Figure 5.4 represents a constraint net example to illustrate the algorithm; it shows a constraint net module. After applying the breadth-first algorithm to transverse the graph from the end to the beginning, together with an out–degree–count in each node, a linear order is reached in figure 5.4, where the numbers denote the order of the node in the breadth-first transverse. However, the order in figure 5.4 is not the final result yet, since the sequence number is calculated with the roots of the output locations instead of the input locations. Therefore in figure 5.4, a correct order is finally acquired after reversing the order in figure 5.4. Based on the final order, the execution of the module works correctly. Although the ordered sequence includes both transductions and locations, the clock only needs to trigger the transductions. The transduction scheduling algorithm, however, does not work without the condition that the constraint net has to be a directed acyclic graph (DAG). Sometimes constraint nets have feedback connections resulting in a few cycles in the graph. In the com-

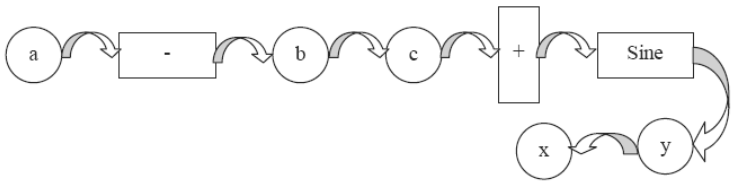


Figure 5.6: (Taken from Song (2002)) The sorted order to fire the transduction.

plex case of a directed cyclic graph, those cycles have to be broken up, and then it becomes an acyclic graph. In PCN modules, a cycle forms when there is a backward connection for creating a feedback. To run the simulation, the particular location in that feedback cycle has to be assigned an initial value (or else, the involved transduction can never get inputs to compute). Such a special kind of location is regarded as a “heuristic tip” for breaking up cycles. When designing a constraint net and confronting a feedback cycle, designers are required to paint the special location in a non-white background color. It also reminds designers to assign an initial value to that location before starting the simulation.

Chapter 6

Concrete Applications of PCNs Framework

This final chapter I describe some concrete applications and problems that are relevant to the research on autonomous robotics. For each problem I propose a PCN–based solution, and furthermore I discuss interesting implications resulting from it. More specifically, I focus on problems arising in two broad areas of robotics: they are (1) behavior–based motor coordination of mobile robots and (2) object recognition and localization for camera–equipped robots.

Robots that use Vision to sense the environment naturally need the ability to recognize objects in the scene. Indeed, assistive robots are supposed to interact actively with the environment and so the further ability to localize and (eventually) reach the objects is crucial too.

Robotic architectures refer to formal models and structures that define a software and hardware framework for controlling robots. They describe the interactions between the components in this framework and provide a structured way for building controllers.

6.1 Several Paradigms for Robot Architecture Design

Mackworth proposed that there are three main research paradigms in robotics, namely Good Old Fashioned AI and Robotics (GOFAIR), Insect AI and Situated Agents, which have

evolved dialectically. GOFAIR utilizes deliberative architectures, Insect AI employs reactive architectures (e.g. subsumption architecture) and Situated Agents advocates the emerging thesis, which is called deliberative/reactive architectures. Characteristics of these paradigms are explained below.

6.1.1 GOFAIR

GOFAIR (e.g. first mobile robots such as Nilsson's Shakey) approach is the first paradigm developed for robotic agent construction. It strongly depends on a set of restrictive assumptions about the agent, the world and interaction between the agent and the world. These assumptions can be listed as follows:

- There is a single agent in the world. Therefore, cooperation between multiple agents is not possible.
- The world can be accurately and completely modelled by the agent and it stays static unless the agent changes it. Hence, the agent does not have the capability to react towards dynamic changes in the world.
- The agent has definite knowledge of everything related to completing its goals and it can predict all the effects of its actions that have been carried out towards reaching these goals. Thus, non-deterministic actions are not supported. Also actions are performed in sequence and concurrent actions are not supported.

Planning constitutes the main activity of the GOFAIR controller. These systems use hierarchical deliberative planners, which have modules that are delegated to clearly identifiable subdivisions of functionality. Modules interact with each other in a predetermined manner and higher modules in the hierarchy provide subgoals for lower modules. Reasoning with rule based manipulation of symbolic structures in the world model is defined as "intelligence" and

sensing and acting in the real world are referred as “secondary concerns”. Sensing is only needed to determine the initial state of the real world in order to construct the world model. Actions that are carried out to reach goals are produced by reasoning in this world model. Since all effects of the actions can be predicted and these predictions can be used to update the world model without percepting the changes in the real world, sensing is not required to maintain the world model. Thus, sensing is not used to produce intelligent actions after the initial model is constructed and it is not directly connected to the acting. However, intelligence in nature is created by the interconnectivity among sensing, reasoning, and acting. Hence, separating them into three distinct modules by assigning importance priorities can not be a scalable approach.

6.1.2 Insect AI

Insect AI (e.g. earlier works of Brooks such as Genghis) paradigm is the antithesis of GOFAIR approach. It advocates reactive architectures. Insect AI does not make an assumption about the world being static and deals with robotic systems that inhabit in environments which are unstructured, dynamic and lack temporal consistency and stability. It uses animal models of behaviour as a basis for construction of these robotic systems, where sensing and acting is tightly coupled to produce realtime responses. Since the actions are produced in a reflexive manner with hardwired reactive motor behaviours, reasoning and the use of world models are minimized and planning is eliminated in these systems. Thus, one important drawback of Insect AI is that it can only produce low-level intelligence, which can be observed in animals of nature such as insects.

Brooks proposed the following principles for reactive behaviour-based robotics:

- **Situatedness**: The robot is a real physical system grounded and embedded in a real world, here and now, acting and reacting in real time. There is a strong two way coupling between the robot and its environment. The world is its own best model and robot

does not operate upon abstract representation of reality but rather real world itself. The robot continuously refers to its sensors rather than to an internal world model.

- **Embodiment**: The robot has a physical body and its interactions with the world cannot be simulated faithfully. The embodiment of robots is critical for two reasons. First, only an embodied robot (not the simulation of it) is fully validated as one that can deal with real world. Second, only through physical grounding can any internal symbolic systems find a place to bottom out and give "meaning" to the processes going on within the system.

- **Intelligence**: Intelligence is determined by the dynamics of interaction of the robot with its world. Simple things to do with sensing and acting in a dynamic environment are necessary basis for high-level intelligence. Therefore, the valid approach for building intelligence is to follow bottom-up model. Consequently, the dynamics of interaction between the robot and its environment are primary determinants of intelligence not the reasoning.

- **Emergence**: It is hard to point a single component as a source of intelligence. Intelligence is not a property of either the agent or the environment in isolation but is rather a result of interplay between them. The way in which the intelligence emerges is described quite differently by GOFAIR and behaviour-based robotics. In GOFAIR, the components of the controller are delegated to "functions" such as sensing, planning, modelling, and learning. The "intelligent behaviour" of the system (e.g. avoiding obstacles, standing up, etc.) emerges from interaction of these "functional" components. However, in behaviour–based robotics, components of the controller are defined as behaviour producing. The "intelligent functionality" of the system (e.g. sensing, learning, etc.) emerges from the interaction of these "behaviour" components.

6.1.3 Situated Agents

Emerging synthesis of GOFAIR and Insect AI is called as Situated Agents (e.g. robots developed by the CBA framework such as Dynamite vehicles). Since they can pursue goals as they react to unpredicted real-time changes in their environments, the robotic controllers designed with this paradigm are both deliberative and reactive. When the uncertainty in the world is restricted and some guarantee is given that no change exist in the world during the execution of the system, the world can accurately be modelled. In these situations, deliberative methods can be used to carry out a complete plan. However, to execute this proactive plan, neither deliberative architectures of GOFAIR nor reactive architectures of Insect AI can be used. Deliberative structures can not be deployed, since all the assumptions of GOFAIR paradigm can not be true in the real world. Reactive counterparts of these architectures can also not be utilized since they do not support planning. Therefore, hybrid deliberative/reactive robotic architectures have emerged under the Situated Agents paradigm. In hybrid architectures, the controller should be able to integrate world knowledge and goals to arrive at a plan prior to execution. It should also be able to respond rapidly and effectively to dynamic changes that occur within its world. Since, the traditional deliberative controllers attempt to pre-plan for all eventualities, they often cause the planning process not to terminate. They commit to arbitrary length plans and do not allow the robot to change its goals in response to unpredictable changes in the world. The reactive approach, on the other hand, is very good at dealing with the immediacy of sensor data but is less effective in integrating world knowledge. Hence, hybrid architectures of Situated Agents paradigm do not center on reactive versus deliberative control but rather on how to synthesize a control regime that incorporates both types of structures. They use symbolic methods and abstract representational knowledge of GOFAIR and maintain the Insect AI's goal of providing the responsiveness, robustness and flexibility. Situated Agent paradigm follows Brooks' principles of reactivity. As in Insect AI, Situated Agent paradigm challenges GOFAIR by grounding the agent in space and

time by proposing tight coupling of sensing and acting. However, it does not follow Insect AI's efforts on reducing reasoning and representation, but rather integrates reasoning with sensing and acting while creating necessary world models. In addition, opposing to Insect AI paradigm, Situated Agents allows planning to be the part of system if needed. However, planning is not the essential activity in Situated Agents as it is in GOFAIR paradigm. Indeed, in this approach, sensing and acting take a preeminence over knowledge representation and planning. Other differences can be listed as follows: Situated Agents approach can have multiple agents in a dynamic world, whereas GOFAIR can only have a single agent in a static world. As in Insect AI paradigm, Situated Agents can operate on unstructured and uncertain environments (e.g. soccer field), whereas GOFAIR is only suitable for structural and highly predictive environments (e.g. manufacturing). Speed of response of the controller increases as we shift from GOFAIR (which mostly uses offline computational models) to Situated Agents (which uses online computational models) and from Situated Agents to Insect AI. However level of intelligence decreases as we shift from GOFAIR to Situated Agents and from Situated Agents to Insect AI.

6.2 Subsumption Architecture

Subsumption is a reactive architecture developed by Brooks which focuses on priority-based arbitration of task-achieving behaviours. Each behavior is represented as a separate layer. Lower levels have no awareness of higher levels and this provides the basis for bottom-up incremental design. The name subsumption arises from this design, where higher level behaviours (e.g. avoid collisions) are added on top of lower level behaviours (e.g. move around) by using priority-based arbitration. Hence, complex behaviours always include simpler behaviours (e.g. in order to avoid collisions the robot should move around). Thus, this architecture allows us to follow the the evolutionary path and to start building simple agents

in the unpredictable real world in order to construct targeted complex systems. Traditional architectures used in GOFAIR paradigm also advocates layered controller structures. However, subsumption architecture and traditional architectures are layered along completely different dimensions as seen in Figure (TO BE INSERTED). GOFAIR architectures use sense-plan-act vertical models where each layer is dedicated to a separate functional unit such as sensing, modelling, planning, and acting. Layers in this model work sequentially and synchronously. The subsumption architecture use a horizontal model where each layer is dedicated to a separate task-achieving behaviour and each behaviour embodies functional units such as sensing and acting. Layers in subsumption work concurrently and asynchronously. Each layer of the subsumption architecture is constituted by networks of augmented finite state machines (AFSM). AFSMs can be defined with a formal model called Behaviour Language which is also developed by Brooks. Finite state machines in AFSM are augmented by timers which enable state changes after predetermined time periods. Reset signals are used to restore behaviour to its original state. Each AFSM encapsulates a particular behavioural transformation function and has an input and output signals in addition to reset signal. Input signals which refer to stimulus of the behaviour can be suppressed and output signals which refers to response of the behaviour can be inhibited by other active behaviours. These mechanisms of suppression and inhibition enforce priority-based arbitration of behaviours and permit communication between layers. However they restrict this communication heavily. The real world itself becomes the primary medium of communication in the following way: Actions taken by one behaviour create changes within the world and at the same time, sensing element of each layer, which reports new perceptions of the world, communicate those changes to the other behaviours. Hence, in subsumption architecture, world models which uses symbolic representations do not exist. Consequently, reasoning and planning activities are not a part of the architecture. Since no representation and reasoning is used, this architecture is called as Şpurelyŧ reactive and only based on a synergy between sensing and acting.

6.3 Object Recognition and Detection

Recognizing objects is one of the fundamental challenges in computer vision. Roughly speaking, the goal of an object recognition (OR) system is to answer the question: *Is **this** (specific) object present in the scene?*. Sometimes, actually, the focus is not on a specific object but rather on the broader class (or category) to which it belongs[1]. In such a case the question is somehow *less* accurate. For example we could be interested in the presence of a car in the scene regardless of its model or color.

As I have said above, sometimes robots need to know *where* the object is (with respect to some reference frame) and not only *if* it is present or not in the scene. Quite surprisingly, not all OR systems are able to provide this last clue[2] and thus it seems useful to distinguish – among them – those that actually do. Henceforth we will call Object Detection (OD) Systems those systems that recognize objects and further locate them into the scene. It is worth saying here that this definition of OD is a bit less standard and (of course) more controversial the one proposed for OR systems. Many authors would agree with it but some could argue that the boundary between the two is quite fuzzy. It is not my concern to defuse such a controversy here but I believe that, for the sake of clarity, it is useful to keep separated the two kinds of systems because there is a huge difference between them from the practical point of view.

In our everyday life we are so many times involved in recognizing objects (or object classes) that the task seems to us straightforward and we definitely underestimate how difficult it is to perform it by means of an artificial, computer vision system.

One of the main difficulty faced by a recognition system is the problem of variability, and the need to generalize across variations in the appearance of either distinct objects belonging to the same class or the same object seen from different views. In fact, We consider an object

[1] Identifying objects as members of a class, such as cars or dogs, is often referred to as *object categorization*, while identifying individuals within the same class is referred to as *object identification*. I adhere to this convention.

[2] See, for example, Serre et al. (2005) in which the feature vector takes account of the image as a *whole* to check if an object is present or not.

to be a part of – or token in – a sensory signal. The precise representation of the object within the signal can undergo changes such as scaling, translation, or other deformations, or it can be contaminated by noise or be partially occluded. These changes give rise to an entire collection or class of signals which can all still be associated with the original object.

6.4 Algorithms for Object Detection and Recognition

The number of new papers and algorithms for Object Recognition proposed yearly in the Computer Vision community is definitely huge. An exhaustive review of all of them is quite impractical. However it is possible and very helpful indeed to try to classify them according to the *conceptual* approach they are based on. Such a categorization will make it easier for us to see the pros and cons of each method and thus to focus on the one that most likely could fit in with our system.

The first, broader split is definitely the one between *appearance–based* and *feature–based* methods. Each of them can be further divided into two separate groups that we call *global* and *local*. This latter split takes account of the information used to decide if the object is present in the scene or not. Loosely speaking, global methods have a holistic approach to classification while local ones aims at detecting (almost independently) smaller components of the objects first. The specific way of building the "model" fo the object leads us to a final methodological differentiation that is somehow independent from and indeed overlays the previous ones.

The "visual" appearance of an object in an image stands for the combined effects of its shape, reflectance properties, pose in the scene, and illumination conditions. Thus it seems quite plausible to describe and characterize the objects by their global appearance and this can be done – at least in principle – by means of an exemplar of how the object should look like in the scene. In the simplest cases the model is simply built from one or more (entire) images that contain the object in the foreground; this technique is called *example–driven* and

among others has been successfully deployed in (Papageorgiou and Poggio 2000, Sung and Poggio 1998)[3]. Given the exemplar, the most popular strategy is based on representing it as point in a high-dimensional space, and then performing some partitioning of the space into regions corresponding to the different objects or object classes. In order to partition the space a variety of methods have been used: the most common ones are nearest–neighbor classification, vector projection to the nearest manifold (Murase and Nayar 1995), feed–forward neural networks (Sahambi and Khorasani 2003) or support vector machines (dos Santos and Gomes 2002).

Alternatives to the "global" approaches can be found among those methods that attempt to describe all object views belonging to the same class using a collection of some basic building blocks – a kind of local appearance descriptors – by extracting several local image patches.

Over the years, researchers' feelings about appearance-based approaches have had highs and lows. Indeed, the approach is conceptually simple and has led to a variety of successful applications, e.g., illumination planning, visual positioning and tracking of robot manipulators, visual inspection and human face recognition. Nevertheless, these methods are not robust to occlusion and suffer from a lack of invariance to scale, rotation and – of course – changes in the viewpoint. Moreover, the high–dimensionality of the representation is a final problem not easy to overcome if one wishes to use many of the standard learning techniques for pattern recognition.

In order to (partly) overcome this last severe weakness of the appearance–based methods people switched to a more *parsimonious* representation by means of the so called principal–component methods. In this methods, a collection of objects within a class – for example a set of faces, cars or bikes – are used and described as a set of (grey-level) N–dimensional vectors, and the principle components of the training images are extracted. The principal components are then used as the building blocks for describing new images within the class, using linear

[3]It is worth making it clear that the authors did not use pure appearance–based description, though.

combination of the basic images. For example, a set of “eigenfaces” is extracted and used to represent a large space of possible faces. In this approach, the building blocks are global rather than local in nature.

As we shall see in the next section, the building blocks selected by our method are intermediate in complexity: they are considerably more complex than simple local features used in previous approaches, but they still correspond to partial rather than global object views.

6.5 The proposed method

In this section I describe an OR schema that overcomes some of the difficulties and limitations mentioned above. The schema comprises several pieces of algorithms and methods that I’ve cited in sec. 6.4. The main novelty of the schema lies in the way the modules hang together. The system is able to recognize (specific views of) an object of interest in a cluttered background and even in presence of partial occlusion. Interestingly enough, it is further able to recognize a specific person – let’s call this person the *instructor* – among several people and, by “looking at” what he holds in his hands, to acquire the model of a new object of interest on the fly. The main asset of this system to socially assistive robotics is the possibility to “ask” a robot to search for any new object (i.e. never seen before)

6.5.1 Face Detection

In order to interact with people, the ability of recognizing faces is crucial. Almost any socially assistive robot might be able – at least – to detect people in the scene very quickly and without too much computational load so that we can design a basic behavior that continuously search for faces popping up in the scene. Such an event can therefore “trigger” a refinement of the recognition and check if the person (just detected) can be ignored or if some kind of interaction is needed. This basic behavior in my system is provided by a module for fast

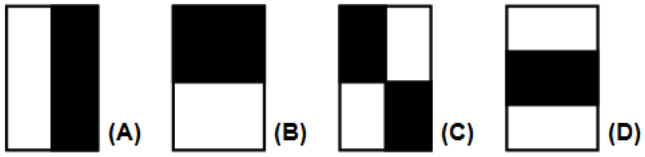


Figure 6.1: Examples of rectangle features. The sum of the pixels which lie within the white rectangles are subtracted from the sum of pixels in the black rectangles. This figure has been adapted from (Viola and Jones 2001)

In this section I give the bare essential of the system for face detection described in (Viola and Jones 2001). Furthermore, I outline how it is possible to define a prior probability $P(body|x,y) = P(body|face,x,y)$ of the pixel (x,y) to belong to the body of a person given that his face is present in a neighborhood.

The module classifies images based on the value of simple features that are strongly reminiscent of Haar basis functions – already used for object recognition by Papageorgiou et al. (1998). The authors describe three different kind of features (see fig. 6.1):

- *two–rectangle* feature is the difference between the sum of the pixels within two rectangular regions with the same size. The regions can be aligned either horizontally (*A*) or vertically (*B*).
- *three–rectangle* feature is the sum of the pixels within a central region subtracted by the sum of the pixels within two regions that are at the opposite sides of the central one (*C*).
- *four–rectangle* feature is the difference between diagonal pairs of rectangles (*D*).

Given a base resolution of the detector, thousands of these simple features can be extracted from each image. However this computation can be speeded up by using the *integral image* trick that involve an intermediate representation for the image[4] that contains, at each location

[4]This intermediate representation is called *Integral Image* in order to emphasize its use for the analysis of the image.

(x, y) the sum of the pixel sabove and to the left of x and y, inclusive. It is easy to show that the sum of all the pixels within a region can be computed by simple addition and subtraction of the value of the corner pixels in the integral image.

Given this set of features, the learning ...(Freund and Schapire 1995, Schapire et al. 1998).

6.5.2 Skin Detection

The part of the system that provides information about where the robot might "look at" in the scene in order to acquire the model of a new object of interest is basically a skin detection module, i.e. it uses skin color as a feature for hand detection. There are two main problems we must face in order to build a robust skin detection module. First, what color space to choose and second, how exactly the skin color distribution should be modelled. Once the skin pixels have been located, a third issue is how to segment the image in order to locate exactly the region where there are the hands that is to say – in our specific case – where the new object of interest is in the image reference frame.

In the literature, we find two main approaches to skin detection: *region–based* and *pixel–based*. According to the former one, spatial arrangement of pixels plays some part in the decision to assign a label to each pixel (Kruppa et al. 2002, Jadynak et al. 2002, Yang and Ahuja 1998). While pixel–based skin detection methods classify each pixel as skin or non-skin individually, independently from its neighbors – see (Vezhnevets et al. 2003) for a complete survey on these latter methods.

In order to develop the module for skin detection I preferred to use the pixel–based approach because it has been shown to be faster than the other one while keeping the performance at high level in a wide range of applications.

The most common camera used in mobile robotics are simple *RGB* color camera and so – for what concern the color space – a quite obvious choice is to work in the standard *RGB*

color space. It is the most widely used color space for processing and storing of digital image data. However, it has two main drawbacks: (1) channels are high correlated between each other, and (2) information about chrominance and luminance is fused together. Thus *RGB* does not seem to be a favorable choice for color based skin detection algorithms. The easiest, obvious candidate as a color space is one obtained from *RGB* by a simple normalization:

$$r = \frac{R}{R+G+B} \quad g = \frac{G}{R+G+B} \quad b = \frac{B}{R+G+B}. \tag{6.1}$$

This normalization leads to two interesting properties. First, the sum of the three components is known ($r + g + b = 1$) and so one of them can be obtained from the other two: we can omit it, reduce the space dimensionality and speed up the computation. Second, it has been shown (Skarbek and Koschan 1994) that normalized *RGB* is invariant (under certain assumptions) of changes of surface orientation relatively to the light surface.

A second, popular color space is the so called *HSV* space that separates out *Hue* (which color it is) from *Saturation* (how concentrated the color is) and *Value*[5] that is tightly related to the brightness of the pixel. Here are the formulas for color conversion from *RGB* to *HSI* (Gonzalez and Woods 2002):

$$\begin{aligned} H &= \arccos \frac{\frac{1}{2}((R-G)+(R-B))}{\sqrt{(R-G)^2+(R-B)(G-B)}} \\ S &= 1 - 3\frac{\min(R,G,B)}{R+G+B} \\ V &= \frac{1}{3}(R+G+B+). \end{aligned} \tag{6.2}$$

The main advantages of using *HSV* color space is that (1) it uses an extremely intuitive manner of specifying color – for instance, it is very easy to select a desired hue and then

[5]Sometimes this last channel is called *Intensity*.

(a) Cylindrical representation (b) Conical representation

Figure 6.2: Two standard visualization methods of the HSV color space. The Cylindrical representation might be considered the most mathematically accurate. However the cone visualization is more practical in most cases because of the limited range of precision of RGB values for digital images.

to modify it slightly by adjustment of its saturation and intensity – and (2) it explicitly discriminates between luminance and chrominance. Thus hue channel, at least in principle, is invariant to surface orientation (relative to the light source) and to highlights at white sources: often ambient light can be considered "approximatively" white. Moreover, good results have been obtained by using only H and S to detect skin pixels. However, there are several undesirable features of this color space that are related to the discontinuities of H and to the fact that, in practice, the number of visually distinct S levels decreases as V approaches zero (see figure 6.2 for more details). Therefore, in the limit $V \to 0$, H becomes quite noisy and useless, since the small number of discrete hue levels cannot adequately represent slight changes in RGB. To overcome this problem, a simple trick could be to ignore pixels that have very low V value. This means that we cannot use the system on very dim scenes. Further, at very low saturation ($S \simeq 0$), variations among H values are tiny and not appreciably different using the usual discrete, 256–levels H scale. Therefore it is a common practise to ignore pixels that have very low S value.

A further color space is the so called YC_rC_b – it is commonly used by European television studios and for image compression work. The space is represented by means of three principal

components: Y (that encode luminance) and two color difference values C_r and C_b that are formed by subtracting luminance from RGB red and blue components.

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ Cr &= R - Y \\ Cb &= B - Y \end{aligned} \tag{6.3}$$

This transformation is easy to compute, yet it explicitly separate luminance and chrominance components. These two motivations make this color space also attractive for skin color modelling.

In order to build a complete skin detection module, the second step is to define a decision rule, that discriminates between skin and non-skin pixels. This is usually accomplished by introducing a metric, which measures distance (in a general sense) of the pixel color to skin model. In the probabilistic framework, the metric is encoded by means of a probability distribution of each pixel – represented by a vector c in the color space – to be either a skin–pixel ($P(skin|c)$) or a non–skin–pixel ($P(\neg skin|c)$). Though it would be nice to assess directly how "correct" it is to assign the label *skin* to each pixel[6], it is not possible to compute $P(skin|c)$ directly from the data. Instead, we can compute how likely is for a color value c to be classified as skin or not, that is to compute $P(c|skin)$. These two quantities can be related by means of the Bayes rule:

$$P(skin|c) = \frac{P(c|skin)P(skin)}{P(c|skin)P(skin) + P(c|\neg skin)P(\neg skin)}. \tag{6.4}$$

From this equation, a simple decision rule can be constructed by introducing a threshold

[6] Actually, this is exactly the meaning of $P(skin|c)$, since it tells us the probability of observing skin given a concrete c color value.

Θ so that we label as skin all those pixels c that satisfy the inequality: $P(skin|c) \geq \Theta$ (Jones and Rehg 1999). However this requires us to know the two priors $P(skin)$ and $P(\neg skin)$ and compute the normalizing factor. It is possible to avoid that and define a new decision rule by means of the ratio of $P(skin|c)$ to $P(\neg skin|c)$, that is:

$$\frac{P(skin|c)}{P(\neg skin|c)} = \frac{P(c|skin)(1 - P(skin))}{P(c|\neg skin)P(skin)} \propto \frac{P(c|skin)}{P(c|\neg skin)} \tag{6.5}$$

It has been possible to get rid of $P(skin)$ in the equation because it doesn't depend on each pixel value and can be taken into account only during the choice of a fixed absolute threshold.

The several methods proposed in literature differ from one another in the way they compute $P(c|skin)$ from the data. The most straightforward methods are the non parametric ones (Birchfield 1998, Sigal et al. 2000, Soriano et al. 2000), which use a histogram based approach. The color space[7] is quantized into a number of bins corresponding to particular range of color[8]. Each bin stores the number of times this particular color occurred in the training skin (and non skin) images. After training, the histogram counts are normalized, converting histogram values to discrete probability distribution. Two clear advantages of the these methods are: (1) they are theoretically independent to the shape of skin distribution in the color space and (2) they are fast in training and usage. This last property is quite appealing in the robotic context we going go use this module. On the opposite side, the main drawback is their inability to interpolate or generalize the training data. Moreover, they require much storage space to store the LUTs; sometimes, in order to reduce the amount of needed memory and to account for possible training data sparsity, coarser color space samplings are used – $128 \times 128 \times 128$, $64 \times 64 \times 64$ and $32 \times 32 \times 32$[9].

The need for more compact skin model representation along with ability to generalize and

[7] Usually, the authors use the chrominance plane only.

[8] In literature, 2D or 3D histograms are referred to as lookup tables (LUT).

[9] The evaluation of different RGB samplings in (Jones and Rehg 1999) has shown, that $32 \times 32 \times 32$ shows the best performance.

interpolate the training data stimulates the development of parametric skin distribution models. I considered only the most populare of them (the Mixture of Gaussians modelling) approach that is able to describe quite well also complex–shaped distributions. The parametrized pdf is:

$$P(c|skin) = \sum_{i=1}^{K} \alpha_i \exp\left\{-\frac{1}{2}(c-\mu_i)^T \Sigma_i^{-1}(c-\mu_i)\right\}, \tag{6.6}$$

where, μ_i and Σ_i are, respectively, the mean vector and covariance matrix of each gaussian in the (either $2D$ or $3D$) color space, K is the number of mixture components and α_i are the so–called mixing parameters that obeys to the normalization constraint: $\sum_i \alpha_i = 1$. Model training can be effectively performed with the Expectation Maximization (EM) algorithm (Yang and Ahuja 1999, Terrillon et al. 2000). The number K of components must be chosen taking into account that the model needs to explain the training data reasonably well with the given model on one hand, and avoid data over-fitting on the other. In the literature, choices range from $K = 2$ to $K = 16$, but a less number of components is preferable because it allow a faster model learning stage with less training samples.

In the chapter **??** I present experimental results obtained with each of the three previous color spaces and both the classification techniques. By taking into account both performance and results I decided to keep *WHICH ONE???* in the complete system.

6.5.3 Sift Features Extraction and Robust Matching

The purpose of this assignment is to learn how to perform object recognition and image matching using local invariant features. The family of features I used is the one based on the SIFT approach described in (Lowe 2004).

In order to compute the features and find the matches among points for each image pairs, I used the pre–compiled binary detector and the simple Matlab matching program provided

on David's home page. The matching algorithm can read lists of keypoints and match them between images.

Figure **??** shows the output of the matching produced by the software. Basically, it extract features from each of the two input images calling the binary detector and then draws lines between features that have close matches.

The key idea behind the program is very simple: the best candidate match for each keypoint is found by identifying its nearest neighbor in the list of keypoints of the second image. The nearest neighbor is defined as the keypoint with minimum Euclidean distance for the invariant descriptor vector. For efficiency in Matlab, it is cheaper to compute dot products between unit vectors rather than Euclidean distances[10].

It is very likely that many features from an image won't have any correct match in the other one because they correspond to background clutter or occlusion or more simply because they are detected in one image only. Therefore, it is necessary to define a way to detect and discard features that do not have any good match in the other list. A naive approach could be to define a global threshold of the neighbor distance, but it can be easily proven that this method is not reliable at all. In (Lowe 2004), again, Lowe proposes to look at the comparisons between the distance of the closest neighbor to that of the second–closest one. The rational behind this approach is the following: we expect that a correct match is highly distinctive and so no keypoint (other than the closest one) should be too close to it. This approach works quite well since, generally, many matches are found and only a small fraction are incorrect. However, as shown in figure **??**, some image pairs could be very difficult to match and a more sophisticated approach is needed.

In order to obtain better results, we should consider what kind of information can help us to assess the correctness of a candidate match. We could use that information to design a *validity checking* for each match and discard only those that do not satisfy the criteria. This

[10]The ratio of angles is a close approximation to the ratio of Euclidean distances for small angles. In the Matlab function the *acos* of dot products is computed and the result is simply used as distance measure.

cascade approach should help us to improve the precision while keeping as high as possible the recall of the system since we only try to filter out wrong matches (decrease the number of false positive).

The key observation that can provide us with a suitable approach is that correct matches will usually have other nearby features vectors that provide consistent matches, while incorrect matches will usually not be consistent with their neighbors.

This idea can be translated easily in an algorithm indeed. For each match from the first image to the second, we should check the N other closest matched features in the first image and check that at least K of them provide locally consistent matches. Unlikely from the above approach, here *close* is related to the image distance.

The last *brick* in this *building* is a compact and effective definition of *local consistency* among matches. Our feature vector, now, are the 4–dimensional vectors: $\mathbf{k} = (R, C, \theta, \sigma)$ where R and C are the row and the column respectively of the keypoint in the image. θ is the orientation in radians and σ is the scale. The main problem with the $\mathbf{k}$ vectors is that they can non be compared homogeneously. For instance, it is worth measuring the similarity between to angles in radians as the difference of them. If we consider the scale, though, it could be more useful to compute the ratio. So, for example, if two keypoints in the first match have an orientation difference of $\Delta\theta$ radians, then the consistent matches should also have an orientation difference close to $\Delta\theta$. However, the ratios of scales (and not the differences) for each match should be similar too. The last cue for a good *local consistency* measure, comes form the spatial proximity of keypoints: two keypoints in the first list that are close each other must have matches that are close too. A simple comparison of these two distances, however, will fail miserably since the occurrence of the object in the second image con have different scale as well: that's one of the strength point of SIFT features and we don't want to loose it, of course.

Given the previous considerations, I propose to use an *ad–hoc* the similarity measure for

keypoints within the same list *i*:

$$\delta_{i_{12}} = \delta_i(\mathbf{k}_{i_1} - \mathbf{k}_{i_2}) = \left[\frac{R_{i_1} - R_{i_2}}{\sigma_{i_1}}, \frac{C_{i_1} - C_{i_2}}{\sigma_{i_1}}, \theta_{i_1} - \theta_{i_2}, \frac{\sigma_{i_1}}{\sigma_{i_2}} \right] \tag{6.7}$$

Definition in Eq. 6.7 is not a Euclidean metric and it is not a metric at all, actually. It is, instead, a kind of similarity vector that we can use, effectively, to assess how close two keypoints are.

Two keypoints *k* and *l*, within the same neighborhood in the list 1 and with a similarity vector $\delta_{1_{kl}}$, are said to be consistent if their corresponding matches in the list 2 have a similarity vector $\delta_{2_{kl}}$ such that:

$$\delta_{1_{kl}} - \delta_{2_{kl}} \leq \textbf{alpha} \tag{6.8}$$

Where α is a threshold vector specified by the user.

The algorithm

The pseudo–code of the algorithm is the described in program 1:

6.5.4 Object Recognition

In this section I describe a probabilistic recognition method that detects an instance of a specific object in the scene. The method relies on finding highly probable matches from SIFT feature vectors extracted in the image of the scene to those extracted from a a snapshot of the object.

Thus the first hypothesis is that we can build a model $\mathcal{M}_O$ of the object O we wish to find and that the model is represented by a list of keypoints extracted by an exemplar of how the object should look like. Even if this method depends strongly on the appearance of the object, actual information is not the image itself – as for appearance–based methods – but instead is

Program 1 Match Validity Check

INPUT:
L_1, L_2: keypoint list of the first and second image respectively;
$\mathcal{M}$: list of candidate matches;
N: size of neighborhood;
K: minimum number of consistent matches in each neighborhood;
alpha: threshold vector;
OUTPUT:
$\mathcal{M}_1$: list of matches that pass the validity check;

create a copy, $\mathcal{M}_1$, of $\mathcal{M}$;
for each $\mathbf{k}_{1_i} \in L_1$
$\mathcal{N}_i \leftarrow N$ nearest neighbors of $\mathbf{k}_{1_i}$ in L_1;
$k := 0$;
$\mathbf{k}_{1_j} \leftarrow$ the nearest neighbor in $\mathcal{N}_i$;
while ($k < K$) AND (there are still vectors left in $\mathcal{N}_i$)
$\mathbf{k}_{2_i}$ = the candidate match in L_2 of $\mathbf{k}_{1_i}$
$\mathbf{k}_{2_j}$ = the candidate match in L_2 of $\mathbf{k}_{1_j}$
$\delta_{1_{ij}(1)} = \frac{|R_{1_i} - R_{1_j}|}{\sigma_{1_i}}$; $\delta_{1_{ij}(2)} = \frac{|C_{1_i} - C_{1_j}|}{\sigma_{1_i}}$; $\delta_{1_{ij}(3)} = |\theta_{1_i} - \theta_{1_j}|$; $\delta_{1_{ij}(4)} = \frac{\sigma_{1_i}}{\sigma_{1_j}}$;
$\delta_{2_{ij}(1)} = \frac{|R_{2_i} - R_{2_j}|}{\sigma_{2_i}}$; $\delta_{2_{ij}(2)} = \frac{|C_{2_i} - C_{2_j}|}{\sigma_{2_i}}$; $\delta_{2_{ij}(3)} = |\theta_{2_i} - \theta_{2_j}|$; $\delta_{2_{ij}(4)} = \frac{\sigma_{2_i}}{\sigma_{2_j}}$;

$\mathbf{\Delta} = |\delta_{1_{ij}} - \delta_{2_{ij}}|$;
if $\mathbf{\Delta} \leq \alpha$
$k = k + 1$;
end if
$\mathbf{k}_{1_j} \leftarrow$ the next nearest neighbor in $\mathcal{N}_i$;
end while
if $k < K$
Discard the match for $\mathcal{M}_1$
end if
end for
return $\mathcal{M}_1$

built upon a number of local features that are coded in the SIFT space. More formally, the model $\mathcal{M}$ is a set:

$$\mathcal{M}_O = \left\{\mathbf{k}_j^O\right\}_{j \in J} \tag{6.9}$$

where J is an index set whose cardinality M depends on how many keypoints have been found in the exemplar image. Similarly we can describe each image over time by means of a set[11]:

$$\mathcal{I} = \{\mathbf{k}_i\}_{i \in I} \, . \tag{6.10}$$

Given this representation approach for both the model and the images of the scene, we can state the problem of finding an instance of the object in the scene as follows:

Definition 6.1 (Recognition Problem Statement) *For each image $\mathbb{I}$, find a subset $O \subseteq \mathcal{I}$ of the list of keypoints whose elements match most likely one corresponding keypoint in the list $\mathcal{M}_O$.*

That is to say find all the candidate subparts of the occurrence of the object in the scene. Once all candidate subparts are detected we check if they are consistent. Only those keypoints that pass the check can give us information about object position and orientation over the image plane and its scale factor with respect to the model. In order to make these considerations more formal and to translate them in an algorithmic form, I propose the following work flow to solve the Recognition Problem 6.1. For each image $\mathbb{I}$:

1. extract the keypoints using the technique described in section **??** and associate a descriptor to each keypoint;

2. assign a measure to each keypoint $\mathbf{k}_i$; we call $P_{\mathcal{M}_O}(\mathbf{k}_i) = P\left(\mathbf{k}_i | \mathbf{k}^O_{1:M}\right) P\left(\mathbf{k}^O_{1:M}\right)$ this measure and it is the probability of $\mathbf{k}_i$ to represent a part of the object O. Such a probability is the product of the likelihood term $P\left(\mathbf{k}_i | \mathbf{k}^O_{1:M}\right)$ – that assess how likely it is to find $\mathbf{k}_i$ in the image given the model – and a prior $P\left(\mathbf{k}^O_{1:M}\right)$ over the model keypoints[12].

[11] For the moment we can omit to mention explicitly the time in the notation.
[12] In section **??** I describe a reasonable choice of the priors.

3. asses the presence of the object O by means of a probability measure $P(O \text{ is present } |\mathbb{I}) = P_{\mathcal{M}_O}(\mathbf{k}_{1:N})$ that is a function of all N the candidate keypoints extracted from $\mathbb{I}$.

4. (if the object is present) compute the mean values of position, orientation and scale of $\mathbf{k}_1 : N$ and assign them to the object.

The proposed method is fairly straightforward yet it is quite robust as you can see from the experimental results presented in chapter **??**. However we still miss three main ingredients: the explicit form of the probability measures and a decision rule to assess the presence of the object in the scene. We can introduce them by using the following arguments.

Recall from section **??** that we discard all those keypoints whose distance from the closest $\mathbf{k}_j^O \in \mathcal{M}_O$ is much the same of that from the second–closest one. This criterion leads to a robust keypoint detector (Lowe 2004), so as a first instance I suggest to use a similar measure to define a probability over the keypoints extracted from each image. Let's call $d_{ij} = \|\Delta(\mathbf{k}_i) - \Delta(\mathbf{k}_j^O)\|$, with $i \in I$ and $j \in J$, the distance between the descriptor vectors associated to each keypoint. Then we can define $d_i' = min\left\{d_{ij}\right\}_{i\in I, j\in J}$ and $d_i'' = min\left(\left\{d_{ij}\right\}_{i\in I, j\in J}\right)/d_i'$ and introduce the likelihood measure:

$$P\left(\mathbf{k}_i|\mathbf{k}_{1:M}^O\right) = \exp\left[-\left(\frac{d_i'}{d_i''}\right)^2\right] \tag{6.11}$$

Chapter 7

Conclusions

Throughout this thesis I adhere to a schematic picture of a robotic system as a combination of a *controller* and a *body* immersed in their *environment* (see figure 1.1). This is the insightful picture that underlies the *Probabilistic Constraint Nets* framework introduced by Robert St–Aubin (St-Aubin 2005, St-Aubin et al. 2006) for the modelling and simulation of stochastic hybrid dynamical system. Formal syntax and semantics are provided for PCNs in order to assess the correctness of the models. The framework comprises a specification language (average timed ∀–automata) and some verification algorithms too; they allow for the formal specification of behavioural constraints on the system and enable us to make on–average/probabilistic verification of the requirements. If one adopts this approach to design robotic architectures, then the *behavior* of a robotic system is defined as the set of observed robot/environment traces. Thus, any behavioral requirement can be formally specified as a subset of all the possible traces[1].

In short, the *formal methods for robotics* proposed by this thesis are based on the following guidelines. Given some requirement specifications[2], one should model the body $\mathcal{B}$ and the environment $\mathcal{E}$, and then build a suitable controller C, such that the behavior of the resulting system satisfies the requirement, that is to say, it verifies the fundamental equation:

$$[\![X = \mathcal{B}(U, Y), U = C(X, Y), Y = \mathcal{E}(X)]\!] \models \mathcal{R}$$

Therefore – at least in principle – within the PCNs framework it is always possible to provide formal guarantees that the robot meets (or does not) the specified requirements.

[1] That is to say the subset of all the traces that satisfy the given property/constraint.

[2] For example *goal achievement* $\mathcal{R}$, *safety guarantee*, and *bounded response* to unexpected events.

This is a valuable asset to the robotic research context, because it is a first concrete answer to the growing need for a common, comprehensive framework for modelling autonomous robots. As extensively discussed in the introduction, this is well motivated by the recent trend in building robots that interact autonomously with people, and even assist disabled people through social interaction.

The contributions of the present thesis to the PCNs framework are threefold. First, in chapter 4, I discuss the relationships between PCNs and several deterministic/probabilistic modeling frameworks commonly used in Robotics. More specifically I consider Artificial Neural Networks, Continuous Time Recurrent Neural Networks, Markov Chains and Markov Processes, Reinforcement Learning systems and Markov Decision Processes, and finally, Bayesian Filtering and Kalman Filters. I show that they can be considered as special cases of the PCNs framework, by providing – for each model – the PCN that actually preserves the semantics of the computation – i.e. the propose PCN computes exactly the same thing.

Second, in chapter 5, I describe an integrated programming environment called PCNJ – that stands for *Probabilistic Constraint Nets in Java* – which supports probabilistic constraint net modelling, simulation, and animation for any kind of hybrid systems. I co–developed PCNJ with Alan Mackworth and Lee Leif Chang from the University of British Columbia (Vancouver B.C.,CA).

Third, in chapter 6, I discuss some concrete applications and problems that are relevant to the research on autonomous robotics. For each problem I propose a PCN–based solution, and furthermore I discuss interesting implications resulting from it. More specifically, I focus on problems arising in two broad areas of robotics: they are (1) behavior–based motor coordination of mobile robots and (2) object recognition and localization for camera–equipped robots.

Bibliography

Asimov, I.: 1942, Runaround, *Astounding Stories* **March**.

Baader, F.: 1996, A Formal Definition for the Expressive Power of Terminological Knowledge Representation Languages, *Journal of Logic and Computation* **6**, 33–54.

Billingsley, P.: 1986, *Probability and Measure*, Wiley series in probability and mathematical statistics.

Birchfield, S.: 1998, Elliptical head tracking using intensity gradients and color histograms, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR 98)*, pp. 232–237.

Breiman, L.: 1968, *Probability*, Addison–Wesley.

Burnett, M. M.: 1999, *Wiley Encyclopedia of Electrical and Electronics Engineering*, John Wiley and Sons, Inc, chapter Visual Programming, pp. 275–283.

Chang, S.: 1990, *Principles of Visual Programming Systems.*, Prentice Hall, New York.

dos Santos, E. and Gomes, H.: 2002, A Comparative Study of Polynomial Kernel SVM Applied to Appearance–Based Object Recognition, *in* S.-W. Lee and A. Verri (eds), *SVM 2002, LNCS 2388*, Springer Verlag, Berlin Heidelberg 2002.

Freund, Y. and Schapire, R.: 1995, A decision–theoretic generalization of on–line learning and an application to boosting, *Computational Learning Theory: Eurocolt 95*.

Gemignani, M. C.: 1967, *Elementary Topology*, Addison–Wesley.

Gonzalez, R. and Woods, R.: 2002, *Digital Image Processing, 2nd Edition*, Prentice Hall.

Hallam, J. and Bruyninckx, H.: 2006, An Ontology of Robotics Science, *in* H. I. Christensen (ed.), *European Robotics Symposium 2006, STAR 22*, Springer–Verlag Berlin, Heidelberg, pp. 1–14.

Hennessy, M.: 1988, *Algebraic Theory of Processes*, MIT Press.

Hopfield, J. J.: 1984, Neurons with graded response have collective computational properties like those of two-state neurons., *Proc Natl Acad Sci U S A* **81**(10), 3088–3092.

Jadynak, B., Zheng, H., Daoudi, M. and Barret, D.: 2002, Maximum entropy models for skin detection, *Technical report*, Universite des Sciences et Technologies de Lille, France.

Jagannathan, R.: 1996, *Parallel and Distributed Computer Handbook*, McGraw-Hill, Inc., chapter Dataflow Models.

Jensen, K.: 1981, Coloured Petri Nets and the Invariant-Method., *Theor. Comp. Science 14* pp. 317–336.

Jones, M. and Rehg, J.: 1999, Statistical color models with application to skin detection, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR 99)*, Vol. 1, pp. 274–280.

Kailath, T., Sayed, A. and Hassibi, B.: 2000, *Linear Estimation*, Prentice Hall, Upper Saddle River, NJ USA.

Kalman, R. E.: 1960, A New Approach to Linear Filtering and Prediction Problems, *Transactions of the ASME–Journal of Basic Engineering* **82**(Series D), 35–45.

Kruppa, H., Bauer, M. and Schiele, B.: 2002, Skin patch detection in real-world images, *Annual Symposium for Pattern Recognition of the DAGM 2002, Springer LNCS 2449*, pp. 109–117.

Lankenau, A. and Meyer, O.: 1999, Formal methods in robotics: Fault tree based verification, *Proc. of Quality Week Europe*, Brussels, Belgium.

Leuschen, M. L., Walker, I. D. and Cavallaro, J. R.: 1998, Robot Reliability Through Fuzzy Markov Models, *Proceedings of the annual Reliability and Maintainability Symposium*.

Levesque, H. and Brachman, R. J.: 1987, Expressiveness and tractability in knowledge representation and reasoning, *Computational Intelligence* **3**.

Lewis, R.: 1986, *Optimal Estimation with an Introduction to Stochastic Control Theory*, John Wiley & Sons, Inc.

Logics in Artificial Intelligence: 2004, Vol. Volume 3229/2004 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, chapter Formal Methods in Robotics.

Lowe, D. G.: 2004, Distinctive Image Features from Scale-Invariant Keypoints, *Int. J. Comput. Vision* **60**(2), 91–110.

Mackworth, A. and Zhang, Y.: 2003, A formal approach to agent design: An overview of constraint-based agents, *Constraints* **8**(3).

Maler, O. and Pnueli, A. (eds): 2003, *Proceedings of Hybrid Systems: Computation and Control, 6th International Workshop - HSCC 2003*, Vol. 2623, Lecture Notes in Computer Science. Springer Verlag, Prague, Czech Republic.

Manes, E. and Arbib, M.: 1986, *Algebraic Approaches to Program Semantics*, Springer-Verlag.

Manna, Z. and Pnueli, A.: 1987, Specification and verification of concurrent programs by ∀-automata, *14th Ann. ACM Symp. on Princ. of Progr. Lang.*, pp. 1–12.

Murase, H. and Nayar, S. K.: 1995, Visual learning and recognition of 3-D objects from appearance, *International Journal of Computer Vision* **14**(1), 5–24.

Muyan-Ozcelik, P.: 2004, *Prioritized Constraints in the Design of a Situated Robot*, Master's thesis, The faculty of Graduate Studies (Computer Science) – The University of British Columbia, Vancouver B.C., CA.

Nebel, B.: 1990, Reasoning and Revision in Hybrid Representation Systems, *Lecture Notes in Computer Science* **422**.

Papageorgiou, C., Oren, M. and Poggio, T.: 1998, A general framework for object detection, *Proceedings of International Conference on Computer Vision*.

Papageorgiou, C. and Poggio, T.: 2000, A trainable system for object detection, *International Journal of Computer Vision* **38**(1), 15–33.

Peterson, J. L.: 1981, *Petri Net Theory and the Modeling of Systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Sahambi, H. S. and Khorasani, K.: 2003, A Neural-Network Appearance-Based 3-D Object Recognition Using Independent Component Analysis, *IEEE Transactions on Neural Networks* **14**(1), 138–149.

Schapire, R., Freund, Y., Bartlett, P. and Lee, W.: 1998, Boosting the margin: a new explanation for the effectiveness of voting methods, *Annals of Statistics* **26**(5), 1651–1686.

Serre, T., Wolf, L. and Poggio, T.: 2005, Object Recognition with Features Inspired by Visual Cortex, *CVPR (2)*, pp. 994–1000.

Seward, D. W., Margrave, F. W., Sommerville, I. and Kotonya, G.: 1995, Safe Systems for Mobile Robots, *Proc. Third Safety of Critical Systems Symposium*, Brighton, England (1995).

Sigal, L., Sclaroff, S. and Athitsos, V.: 2000, Estimation and prediction of evolving color distributions for skin segmentation under varying illumination, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2000).*, Vol. 2, pp. 152–159.

Skarbek, W. and Koschan, A.: 1994, Colour image segmentation Ű- a survey, *Technical report*, Institute for Technical Informatics, Technical University of Berlin.

Song, F.: 2002, *CNJ: A Visual Programming Environment For Constraint Nets*, Master's thesis, The faculty of Graduate Studies (Computer Science) – The University of British Columbia, Vancouver B.C., CA.

Sorenson, H.: 1970, Least-squares estimation: from Gauss to Kalman, *IEEE Spectrum* **7**, 63–68.

Soriano, M., Huovinen, S., Martinkauppi, B. and Laaksonen, M.: 2000, Skin detection in video under changing illumination conditions, *Proceedings of 15th International Conference on Pattern Recognition*, Vol. 1, pp. 839–842.

St-Aubin, R., Friedman, J. and Mackworth, A.: 2006, A Formal Mathematical Framework for Modeling Probabilistic Hybrid Systems, *(AI & Math 2006) Ninth International Symposium on Artificial Intelligence and Mathematics*.

St-Aubin, R. J.: 2005, *Probabilistic Constraint Nets: A Formal Framework for the Modeling and Verification of Probabilistic Hybrid Systems.*, PhD thesis, The faculty of Graduate Studies (Computer Science) – The University of British Columbia, Vancouver B.C., CA.

Sung, K. and Poggio, T.: 1998, Example–based learning for view–based human face detection, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **20**(1), 39–51.

Terrillon, J., Shirazi, M., Fukamachi, H. and Akamatsu, S.: 2000, Comparative performance of different skin chrominance models and chrominance spaces for the automatic detection of human faces in color images, *Proceedings of the International Conference on Face and Gesture Recognition*, pp. 54–61.

Tesfagiorgis, B. G.: 2006, *Expressivity of Timed Automata Models*, PhD thesis, Radboud University Nijmegen.

Tomlin, C. and Greenstreet, M. R. (eds): 2002, *Proceedings of Hybrid Systems: Computation and Control, 5th InternationalWorkshop - HSCC 2002*, Vol. 2289, Lecture Notes in Computer Science. Springer Verlag, Stanford,CA,USA.

Vezhnevets, V., Sazonov, V. and Andreeva, A.: 2003, A survey on pixel-based skin color detection techniques, *Proceedings of the Graphicon 2003*, pp. 85–92.

Viola, P. and Jones, M.: 2001, Rapid Object Detection using a Boosted Cascade of Simple Reatures, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2001)*.

Warga, J.: 1972, *Optimal Control of Differential and Functional Equations*, Academic Press, New York.

Welch, G. and Bishop, G.: 2001, An Introduction to the Kalman Filter - TR 95-041, *Technical report*, University of North Carolina at Chapel Hill, Department of Computer Science.

Williams, D.: 1991, *Probability with martingales*, Cambridge Mathematical Textbooks.

Woods, W.: 1983, What's important about knowledge representation, *Computer* **16**.

Yang, M. and Ahuja, N.: 1998, Detecting human faces in color images, *International Conference on Image Processing (ICIP) 1998*, pp. 127–130.

Yang, M. and Ahuja, N.: 1999, Gaussian mixture model for human skin color and its application in image and video databases, *In Proceedings of the SPIE: Conference on Storage and Retrieval for Image and Video Databases*, Vol. 3656, pp. 458–466.

Zhang, Y.: 1994, *A Foundation for the Design and Analysis of Robotic Systems and Behaviors*, PhD thesis, The faculty of Graduate Studies (Computer Science) – The University of British Columbia, Vancouver B.C., CA.

Zhang, Y. and Mackworth, A. K.: 1995, Constraint nets: a semantic model for hybrid systems, *Theoretical Computer Science* **138**(1), 211–239.

Zhang, Y. and Mackworth, A. K.: 1996, Specification and verification of hybrid dynamic systems by timed ∀-autom ata, *in* R. Alur, T. Henzinger and E. Sontag (eds), *Hybrid Systems III. Verification and Control*, number 1066 in *LNCS*, Springer-Verlag.