1
Introduction

Dynamic Logics (Fischer & Ladner, 1979; Harel et al., 2000) are based on the idea proposed by Pratt (1976) of a modal system where each program composes a modality. They are often used for reasoning about programs and Propositional Dynamic Logic (PDL – Fischer & Ladner, 1979) is one of its most well-known variants (detailed in Chapter 2). The logic has a set of basic programs and a set of operators (sequential composition, iteration and nondeterministic choice) that are used to inductively build the set of non-basic programs. PDL has been used to describe and verify properties and behaviour of sequential programs and systems. Correctness, termination, fairness, liveness and equivalence of programs are among the properties that one usually wants to verify. Each program π corresponds to a modality $\langle\pi\rangle$, where a formula $\langle\pi\rangle\alpha$ means that after some running of π, α is true, considering that π halts. There is also the possibility of using $[\pi]\alpha$ (as an abbreviation for $\neg\langle\pi\rangle\neg\alpha$) indicating that the property denoted by α holds after every possible running of π. A Kripke semantics can be provided, with a frame $\mathcal{F} = \langle W, R_\pi\rangle$, where W is a non-empty set of possible program states and, for each program π, R_π is a binary relation on W such that $(s,t) \in R_\pi$ if and only if there is a computation of π starting in s and terminating in t.

There are a lot of variations of PDL for different approaches (Balbiani & Vakarelov, 2003). Among them, Propositional Algorithmic Logic (Mirkowska, 1981) that analizes properties of programs connectives, the interpretation of Deontic Logic as a variant of Dynamic Logic (Meyer, 1987), applications in linguistics (Kracht, 1995), Multi-Dimensional Dynamic Logic (Petkov, 1987) that allows multi-agent (Khosravifar, 2013) representation, Dynamic Arrow Logic (van Benthem, 1994) to deal with transitions in programs, Data Analysis Logic (del Cerro & Orlowska, 1985), Boolean Modal Logic (Gargov & Passy, 1990), logics for reasoning about knowledge (Fagin et al., 2004), logics for knowledge representation (Lenzerini, 1994) and Dynamic Description Logic (Wolter & Zakharyaschev, 2000). Dynamic Logics provide a large amount of systems and tools and has been used in Model Checking (De Giacomo & Massacci, 1998; Göller & Lohrey, 2006; Lange, 2006).

Petri Net is a widely used formalism to specify and to analyze concurrent programs with a very intuitive graphical representation (detailed in Chapter 2). It allows for representing true concurrency and parallelism in a natural way.

We present the logic Petri-PDL (detailed in Chapter 3 and published by Benevides et al., 2011) that replaces the conventional PDL programs by Marked Petri Net programs. So if π is a Petri Net program with markup s, then the formula $\langle s, \pi \rangle \varphi$ means that after some running this program with the initial markup s, φ will be true (also possible a $\Box$-like modality replacing the tags by brackets as an abbreviation for $\neg\langle s, \pi\rangle\neg\varphi$).

As pointed out by the work of Mazurkiewicz (1987, 1989), logics that deal with Petri Nets use to be incomplete. So we restrict this work to a subset where we can achieve decidability and completeness. We call this subset (defined in Chapter 3) normalised Petri Net.

This work falls in the broad category of works that attempt to generalize PDL and build Dynamic Logics that deal with classes of non-regular programs. As examples of other works in this area, we can mention Harel & Raz (1993); Harel & Kaminsky (1999) and Löding et al. (2007), that develop decidable dynamic logics for fragments of the class of context-free programs and Abrahamson (1980); Goldblatt (1992*b*); Peleg (1897, 1987) and Benevides & Schechter (2008), that develop Dynamic Logics for classes of programs with some sort of concurrency. Petri-PDL has a close relation to two logics in this last group: Concurrent PDL (Peleg, 1897) and Concurrent PDL with Channels (Peleg, 1987). Both of these logics are expressive enough to represent interesting properties of communicating concurrent systems. However, neither of them has a simple Kripke semantics. The first has a semantics based on *super-states* and *super-processes* and its satisfiability problem can be proved undecidable. Also, it does not have a complete axiomatization (Peleg, 1987). On the other hand, Petri-PDL has a simple Kripke semantics, simple and complete axiomatization and the finite model property.

There are other approaches that use Dynamic Logic for reasoning about specifications of concurrent systems represented as Petri Nets (Hull, 2005; Hull & Su, 2005; Tuominen, 1990). They differ from this approach by the fact that although they use Dynamic logic as a specification language for representing Petri Nets, they do not encode Petri Nets as programs of a Dynamic Logic. They translate Nets into PDL language while we have a new Dynamic Logic tailored to reasoning about Petri Nets in a more natural way.

But Petri Nets have limitations in their expressability. Take a two processor system with a shared memory where each processor has a different clock. This scenario may be taken as a problem of real time, or as a problem of

productiveness. The latter turns out to be a probabilistic problem. Thus, using probability is a way to add expressivity power to Petri-Nets. The Dynamic Logic derived from this extension seems to be worth of defining and studying.

Random phenomena are ubiquitous to our everyday experience. Weather changing and equipment failures, mostly unpredictable, are familiar to anyone. Real-time and fault-tolerant computer systems have to consider these randomness phenomena and should be designed by taking some environmental parameters into account. The designer of old multi-user backuping system considered the failure probability of the hard-disks logical (tracks and sectors) and physical (wr-heads) components to build a fault-tolerant driver able to provide a quality of service according to the requirements. Of course this can be one of the simplest example in this field of modelling and performance evaluation (the subarea of computer science that studies and develops tools and methods to help modelling and evaluating computer systems subject to work taking randomness phenomena into account).

Stochastic process is a mathematical modelling tool largely used for describing phenomena of a probabilistic nature as a function of time as a mandatory parameter (Marsan, 1990*a*). Taking for free the definition of a probability space and random variable (Kolmogorov, 1956), a stochastic process $\{Y(t) : t \in [0, \infty)\}$ is a family of random variables defined over the same probability space and taking values in the same state space. Thus, a stochastic process can be understood as a family of functions of time that raise *sample paths*, i.e. trajectories in the state space. General stochastic processes can be quite complex. Among them, those that have no memory of the trajectory to reach the present state (mathematically one require that $\Pr(Y(t) \leq y \mid Y(t_n) = y_n \cdots Y(t_0) = y_0) = \Pr(Y(t) \leq y \mid Y(t_n) = y_n)$, for $t > t_n \cdots > t_1 > t_0$), also called Markov processes, have been widely considered for modelling computational processes. Markov processes with a discrete state space are denominated Markov chains. If the time is continuous, the term Continuous Time Markov Chain (CTMC) is commonly used. CTMC are natural models of computing systems considered in environments subject to randomness (internal or external). CMTCs compete with Queueing networks as tools for modelling and performance evaluation. However, the later does not provide clean mechanisms to describe synchronization, blocking and forking (i.e. consumer splitting). On the other hand, Petri Nets are quite good on describing these last mentioned aspects of a system. The proposal of Stochastic Petri Nets (SPN) brought an equilibrium between the modelling and the performance evaluation phases of systems designing. In the work of Marsan et al. (1984) it is proposed SPNs, in order to avoid the translation

of queueing networks defined in the modelling phase into complex CTMCs for the evaluation phase. In Chapter 4 the definition of SPN requires that transitions are enabled according to an exponential probability distribution (in fact a negative exponential distribution). This requirement is essential in order to ensure that the Stochastic process naturally derived from an SPN is a CTMC (Marsan & Chiola, 1987*a*).

There are some other well-known stochastic approaches to PDL, but we believe that the fact that the probabilistic feature present in each of these formalism was added in a non-structured way. We say that a probabilistic formalism has more structure than other, whenever the first has cleaner Markovian structures than the other. In this sense, the system P-Pr(DL) (Feldman, 1983, 1984), which has no finite axiomatization, does not allow boolean combination of propositional variables and is defined only for regular programs is the less structured. Pr(DL) (Feldman & Harel, 1984) which has the same limitations as P-Pr(DL) and is undecidable, compares to the former. The system PPDL (Kozen, 1983) computes the probability of a proposition being true in some state but the program is replaced by a measurable function, that is, its stochastic component is not compositional. Finally, PPDL$> r$ (Tiomkin & Makowsky, 1985) can only describe situations where some probability is greater then a constant $r \in \mathbb{R}$ and PPDL> 0 (Tiomkin & Makowsky, 1991) that can only describe situations where some probability is greater then zero, showing how the parameters of the modelling impose restriction in the queries, reverting completely the role of a model in a formal verification. We present an extension of Petri-PDL to deal with Stochastic Petri Nets in Chapter 4 (with a variation in Chapter 5).

We present Natural Deduction with labels systems for some of the logics and a Resolution based system for Petri-PDL. The conclusions and further work are in Chapter 6.