CHAPTER 1

MODELING AND VERIFICATION OF DISTRIBUTED SYSTEMS USING MARKOV DECISION PROCESSES

Marco Beccuti[1], Giuliana Franceschinis[2] and Jeremy Sproston[1]

[1]Dipartimento di Informatica, Università di Torino, Italy.
{beccuti,sproston}@di.unito.it
[2]DiSIT, Istituto di Informatica, Università del Piemonte Orientale, Italy.
giuliana.franceschinis@di.unipmn.it

Abstract.
The Markov Decision Process (MDP) formalism is a well-known mathematical formalism to study systems with unknown scheduling mechanisms or with transitions whose next-state probability distribution is not known with precision. Analysis methods for MDPs are based generally on the identification of the strategies that maximize (or minimize) a target function based on the MDP's rewards (or costs). Alternatively, formal languages can be defined to express quantitative properties that we want to be ensured by an MDP, including those which extend classical temporal logics with probabilistic operators.

The MDP formalism is low level: to facilitate the representation of complex real-life distributed systems higher-level languages have been proposed. In this chapter we consider Markov Decision Well-formed Nets (MDWN), which are probabilistic extensions of Petri nets that allow one to describe complex nondeterministic (probabilistic) behavior as a composition of simpler nondeterministic (probabilistic) steps, and which inherit the efficient analysis algorithms originally devised for well-formed Petri nets. The features of the formalism and the type of properties that can be studied are illustrated by an example of a peer-to-peer illegal botnet.

Keywords. Markov decision processes, modeling and verification.

1.1 Introduction

The mathematical formalism of Markov Decision Processes (MDPs) was introduced in the 1950s by Bellman and Howard [17, 7] in the context of operations research and dynamic programming, and has been used in a wide area of disciplines including economics, manufacturing, robotics, automated control and communication systems. An MDP can be regarded as a Markov chain extended with nondeterministic choice over actions, and is typically equipped with rewards (or costs) associated with transitions from state to state.

A key notion for MDPs is that of strategy, which defines the choice of action to be taken after any possible time step of the MDP. Analysis methods for MDPs are based on the identification of the strategies which maximize (or minimize) a target function either based on the MDP's rewards (or costs), or based on properties satisfied by the MDP's execution paths. For example, in a distributed system, there may be different recovery and preventive maintenance policies (modeled by different actions in the MDP); we can model the system using an MDP in order to identify the optimal strategy with respect to reliability, e.g., the optimal recovery and preventive maintenance policy that maximizes system availability. Reward-based performance indices rely on standard methods for MDPs, whereas path-based properties rely on probabilistic model checking methods [8, 3].

It is important to observe that the formalism of MDPs is low level, and it could be difficult to represent directly at this level a complex real-life distributed system. To cope with this problem, a number of higher-level formalisms have been proposed in the literature (e.g., stochastic transition systems [13], dynamic decision networks [14], probabilistic extensions of reactive modules [1], Markov decision Petri nets and Markov decision well-formed nets [5], etc.).

In this chapter we introduce the MDP formalism in the context of distributed systems and discuss how to express and compute (quantitative) properties which should be ensured by an MDP model (Sec. 1.2). Markov decision well-formed nets (MDWNs) are presented highlighting how they can be a good choice to model multi-component distributed systems (Sec. 1.3) such as an illegal botnet example. Standard

MDP analysis and probabilistic model checking techniques are used to compute a number of performance indices on the illegal botnet example (Sec. 1.4).

An application example: peer-to-peer botnet. The application example presented in this chapter is inspired by the peer-to-peer illegal botnet model presented in [23]. Illegal botnets are networks of compromised machines under the remote control of an attacker that is able to use the computing power of these compromised machines for different malicious purposes (e.g., e-mail spam, distributed denial-of-service attacks, spyware, scareware, etc.). Typically, infection begins by exploiting web browser vulnerabilities or by involving a specific malware (a Trojan horse) to install malicious code on a target machine. Then the injected malicious code begins its bootstrap process and attempts to join to the botnet. When a machine is connected to the botnet it is called a *bot*, and can be used for a malicious purpose (we say that it becomes a *working bot*) or specifically to infect new machines (it becomes a *propagation bot*). This choice is a crucial aspect for the success of the malicious activity, meaning that the trade-off between the number of working bots and the number of propagation bots should be carefully investigated. To reduce the probability to be detected, the working and propagation bots are inactive most of the time. A machine can only be recovered if an anti-malware software discovers the infection, or if the computer is physically disconnected from the network.

Our MDP model is similar to that of [23], apart from the fact that we let the choice between the type of malicious activity, working or propagating, be nondeterministic, rather than a fixed probabilistic choice. In this way, we represent all possible choices of assignment of activity to an infected machine, including dynamic strategies that adapt their behaviour to the current global state of the botnet. We consider performance indices such as the average number of working or propagation bots at time t, and the probability that the number of working machines exceeds some threshold within time t. The performance indices obtained from our model are often significantly different from those obtained from a purely probabilistic version of the model in which the choices of activity of a newly infected machine have equal probability.

1.2 Markov Decision Processes

Since the aim of this chapter is to describe how dynamic distributed systems can be effectively studied using MDPs, in this section we introduce the MDP formalism, while in the next section we consider a more high-level formalism for the description of systems which are based on MDPs (more precisely, MDPs provide the underlying semantics of the high-level formalism).

An MDP comprises a set of states, which for the purposes of this chapter we can consider as being finite, together a description of the possible transitions among the states. In MDPs the choice as to which transition to take from a state s is made according to two phases: the first phase comprises a nondeterministic choice among a number of *actions* available in the state s; whereas the second phase comprises a probabilistic choice between the possible target states of the transition. The proba-

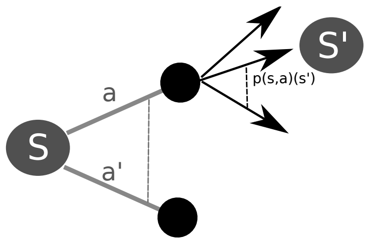


Figure 1.1 A portion of a generic MDP

bility distribution used to choose the next state of the model in the second phase is determined by the choice of action made in the first phase.

The possibility to combine both nondeterministic and probabilistic choice in MDPs is useful in a number of different contexts. In the context of the formal modeling of systems, nondeterministic choice can be used to represent such factors as interleaving between concurrent processes, unknown implementation details, and (automatic or manual) abstraction.

In the following, we use a set of atomic propositions denoted by AP, which will be used to label the states of an MDP. For example, states corresponding to a system error will be labeled with a certain atomic proposition to distinguish them from non-error states.

A discrete probability *distribution* over a finite set Q is a function $\mu : Q \to [0,1]$ such that $\sum_{q \in Q} \mu(q) = 1$. We use $\mathsf{Dist}(Q)$ to denote the set of distributions over Q.

We now formally present the definition of MDPs. An MDP $\mathsf{M} = (S, A, p, r, l)$ comprises: 1) a set S of states; 2) a set A of actions; 3) a partial transition function $p : S \times A \to \mathsf{Dist}(S)$; 4) a partial reward function $r : S \times A \to \mathbb{Q}$; 5) a labeling function $l : S \to 2^{\mathrm{AP}}$.

The transition function, when defined for a state $s \in S$ and action $a \in A$, maps to a distribution $p(s,a)$ over states. For each state $s \in S$, we let A_s denote the set of actions $a \in A$ for which $p(s,a)$ is defined. We assume that A_s is non-empty for each state $s \in S$. Intuitively, when in the state s, the choice of action a is made nondeterministically from A_s, and subsequently the next state will be s' with probability $p(s,a)(s')$ (see Fig. 1.1). The partial reward function is defined for all states $s \in S$ and actions $a \in A_s$, and maps to a rational value $r(s,a)$. The labeling function associates with each state $s \in S$ a set of atomic propositions $l(s)$, which represents a set of observable events that are satisfied in s. The labeling function is used for analyses based on probabilistic model checking, which will be considered in later sections.

There is no unique probability measure associated with an MDP: instead there is a (generally countably infinite) number of probability measures, each of which corresponds to a different way of resolving the nondeterministic choice during the execution of the system. We use the notion of *strategy* (also called schedulers or policies in the literature) to represent a particular selection of all of the possible nondeterministic choices to be made throughout the MDP's execution. A strategy is a function mapping from each possible execution history, representing the behavior of the MDP up to a certain point, to the next action to take. Each strategy defines a Markov chain, on which a probability measure over system executions can be defined in the standard way. We are generally interested in extremal strategies with regard to some criteria regarding performance or correctness: for example, we may wish to consider (optimal) strategies which result in the maximal discounted expected reward in steady-state, or those which result in the minimum probability of system error.

Computing optimal rewards and optimal strategies. The problem of computing a maximal (or minimal) reward and its associated optimal strategy for an MDP can be formulated as a linear program and computed in polynomial time (see, for example, [21]). However linear programming is not practical for large MDPs, and therefore alternative solution techniques based on iterative methods have been proposed, such as value iteration and policy iteration. Value iteration [7] consists in the successive approximation of the required values. At each iteration, a new value for a state is obtained by taking the maximum (or minimum) of values associated with the state's outgoing actions. A value of an action is derived as a weighted sum over the values, computed during the previous iteration, of the possible next states, and where the weights are obtained from the probability distribution associated with the actions.

Each iteration can be performed in time $O(|S|^2 \cdot |A|)$.

In contrast, the policy iteration algorithm proposed by Howard [17] alternates between a *value determination phase*, in which the current policy is evaluated, and a *policy improvement phase*, in which an attempt is made to improve the current computed policy. The policy improvement step can be performed in $O(|S|^2 \cdot |A|)$, while the value determination phase in $O(|S|^3)$ by solving a system of linear equations. Therefore, for both methods, the total running time is polynomial if and only if the number of iterations required to find an optimal policy is polynomial.

Model checking for MDPs. Probabilistic model checking is an extension of model checking, a formal, automatic verification technique which has been used for the analysis of a wide variety of systems [12, 2]. In addition to a formal model of the system, model checking also requires a formal model of the correctness property (or properties) that the system model should ideally satisfy. Correctness properties are typically represented in temporal logic [22, 11], and refer to sequences of system events, such as "a goal state is reached within 100 execution steps" or "a response always follows a request". A model-checking algorithm is executed to establish automatically whether the system model satisfies the property. Model checking has been extended to the case of probabilistic systems, including MDPs [8, 3]. Properties are specified in a probabilistic extensions of classical temporal logic, and refer to the maximum ot minimum probability of temporal logic properties over execution sequences; for example, a system may be regarded as correct if the maximum probability of reaching a goal state within 100 steps is greater than 0.9. Model checking for MDPs relies on the combination of the aforementioned techniques for computing optimal rewards and strategies together with techniques from the field of (non-probabilistic) model checking. A comprehensive overview of probabilistic model checking for MDPs can be found in [15].

1.3 Markov Decision Well-Formed Net formalism

The MDP formalism is rather low level, since the system evolution must be expressed by explicitly describing all possible states, actions and corresponding probabilistic transitions. A higher-level (possibly domain-specific) description can ease the task of the modeler and reduce the risk of introducing errors. In this chapter, we concentrate on the MDWN formalism [5]. We introduce MDWNs in this section, assuming that the reader is familiar with the basic concepts on Well-Formed Nets (WNs, the details of which can be found in [10] or in Appendix A). The formalism is designed to ease the representation of multi-component (distributed) systems: a set C_0 of component identifiers must always be defined in a MDWN model.

An MDWN model is composed by two WN subnets, the probabilistic subnet N^{pr} (enriched with a transition weight function) and the nondeterministic subnet N^{nd}: these submodels represent respectively the probabilistic and nondeterministic behavior of an underlying MDP. The two submodels share the same set of color classes $\mathcal{C}$ (including the special color class C_0 comprising controllable and not controllable components) and the same set of places P. The sets of transitions are instead disjoint $T^{pr} \cap T^{nd} = \emptyset$, and are partitioned into two subsets, *run* and *stop*. Each transition is associated with a subset of components; for the transitions in T^{nd} these must be controllable components.

Transitions of WNs represent parametric events: the transition parameters are typed variables (where the possible types are the color classes in $\mathcal{C}$) appearing on the arcs connected to the transition. A *transition instance* is characterized by a binding (assigning a color from the appropriate class to each variable). In MDWN models (some of) the transition variables represent parametric components ($compvar$); hence a transition instance corresponds to an event where one component or a set of components are involved. If C_0 has two or more static subclasses, each component variable of each transition may be constrained to belong to one static subclass (or to the union of some static subclasses).

The formal definition of MDWN now follows. A Markov Decision Well-Formed Net is a tuple $\mathcal{N}_{MDWN} = \langle N^{pr}, N^{nd}, compvar \rangle$, where:

- $N^{pr} = \langle P, T^{pr}, \mathcal{C}, cd_P \cup cd_T^{pr}, I^{pr}, O^{pr}, H^{pr}, \phi^{pr}, prio^{pr}, weight^{pr}, m_0 \rangle$ is a WN with weights associated with the transitions ($weight^{pr}(t) : cd(t) \to \mathbb{Q}$);
- $N^{nd} = \langle P, T^{nd}, \mathcal{C}, cd_P \cup cd_T^{nd}, I^{nd}, O^{nd}, H^{nd}, \phi^{nd}, prio^{nd}, m_0 \rangle$ is a WN;
- let $t \in T^{pr} \cup T^{nd}$ and $cd_T(t) = var_1 : type_1, \ldots, var_n : type_n$; $compvar(t)$ is a subset of variables in $cd_T(t)$, all of type C_0, used to specify which components are involved in each instance of t. If $|compvar(t)| > 1$, the transition guard must ensure that the same component cannot be assigned to more than one variable. If the component corresponding to a given variable should belong to a specific static subclass of C_0, the transition guard must imply this constraint.

An MDWN model must also have an associated reward function, used to compute the corresponding MDP reward to be optimized: it is defined as a 3-tuple $\langle rs, rt, rg \rangle$ where:

- $rs : \bigotimes_{p \in P} \mathbb{N}^{\widetilde{cd}(p)} \to \mathbb{Q}$, where $\widetilde{cd}(p)$ represents the projection of colors in $cd(p)$ on its *static partition*. In other words, if $c \in cd(p)$ is a tuple of colors from $cd(p)$, $\widetilde{c}$ is the tuple obtained by substituting its elements with the identifier of the static subclass they belong to.

- rt is a T^{nd}-indexed function $rt(t) : \widetilde{cd}(t) \to \mathbb{Q}$ defining a reward value for each occurrence of a given element of the static partition of $cd(t)$;

- $rg : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$ is the function used to combine the values from rs and rt in a unique global reward value: for any given state s and action σ the reward is defined as $rg(rs(s), \sum_{(t,c) \in \sigma} rt(t)(\tilde{c}))$.

The dynamics of an MDWN model is defined as follows: starting from the initial state s_0 the nondeterministic subnet evolves until exactly one stop transition has fired for each controllable component: this kind of trace is called *maximal nondeterministic transition sequence* (MNDTS), corresponding to a (composite) action a, and leading the system to an intermediate marking $m_{s_0,a}$ where the probabilistic subnet must evolve. From such a state a *maximal probabilistic transition sequence* (MPTS) is fired, containing exactly one stop transition for each component in C_0. A probability can be computed on any MPTS based on the function $weight^{pr}$. Then the two steps are repeated, starting from the state reached by the MPTS. This behavior can be obtained by proper composition of the two subnets (see [5] for the details), and application of a standard Reachability Graph (RG) construction algorithm to the complete model. The MDP can be obtained from the RG as follows: the set of states S comprises the initial state and all the states in the RG reached by some MPTS. The actions associated with state s are $A_s = \{t\_count(\sigma) : s \xrightarrow{\sigma}\}$ where σ is a MNDTS that can be fired in s, and $t\_count(\sigma)$ is a T^{nd}-indexed vector of natural numbers, counting how many times each transition instance fires in σ. For each $a \in A_s$ the transition probability distribution corresponding to $m_{s,a}$ (the marking obtained firing all the transitions of the MNDTS represented by a from state s) is obtained as follows: $p(m_{s,a})(s') = \sum_{\sigma' : m_{s,a} \xrightarrow{\sigma'} s'} Pr(\sigma')$ where $m_{s,a} \xrightarrow{\sigma'} s'$ denotes that σ' is a MPTS that can fire from $m_{s,a}$ reaching s', and $Pr(\sigma')$ is obtained as the product of the probabilities of each transition instance appearing in σ', computed according to function $weight^{pr}$. The set of possible actions in each state s may be reduced in case there exist two (or more) different actions a and a' such that $m_{s,a} = m_{s,a'}$: in this case only the action with optimal reward (maximal or minimal, depending whether the reward should be maximized or minimized) is kept, assuming that the $rg()$ function be monotone with respect to its second argument.

Due to the symmetric properties of the MDWN arc functions, guards, weights and reward, the RG (and MDP) size can be considerably reduced by automatically aggregating equivalent states and transitions. As for WNs, the aggregate RG can be generated directly, without first building the whole RG. In the next section an MDWN model of the peer-to-peer botnet is presented and studied.

We now consider briefly an alternative high-level formalisms for MDPs. Models of the probabilistic model checking tool PRISM [20] consist of a number of modules, each of which corresponds to a number of transitions. Each transition is guarded by a condition on the model's variables, and the transitions of a module can update local variables of the module. Multiple transitions may be simultaneously enabled, and the choice between them is nondeterministic; the chosen transition determines a probabilistic choice as to how the variables should be updated. Modules may communicate through synchronization on shared actions with other modules. PRISM does not directly support a multistep nondeterministic or probabilistic transition accounting for the evolution of all components in a given time unit: this can be explicitly modeled by using a variable for each component which records whether the component has taken a transition this time unit.

We briefly mention other formalisms for MDPs. The modeling language MODEST [9] incorporates aspects from process modeling languages and process algebras, and includes MDPs as one of the many formalisms which it can express. Stochastic transition systems [13] also subsume MDPs, but also permit both exponentially timed and immediate transitions. A number of process algebras featuring nondeterministic and probabilistic choice have been introduced; we refer to [18] for an overview of a number of these.

1.4 Case study: Peer-to-Peer Botnets

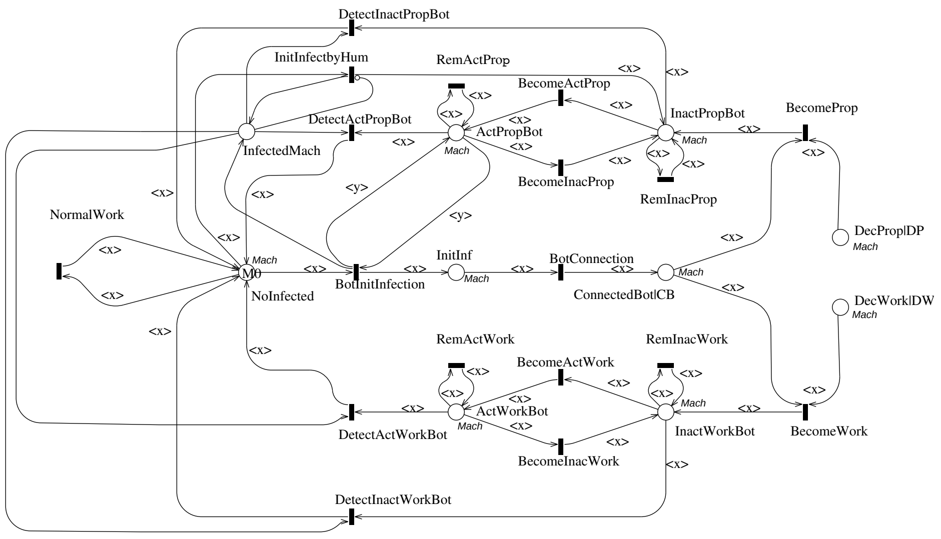
In this section the MDWN model of the peer-to-peer botnet example is described and its quantitative properties are assessed using different analysis techniques.

The probabilistic and nondeterministic subnets of the botnet model are shown in Figs. 1.2(a) and (b) respectively. In this model $\mathcal{C}$ contains only one color class *Mach* representing the computers that can potentially be infected (initially all in place *NoInfected*). This is also the color class representing the components of the model, which are all controllable. All transitions both in the probabilistic and in the nondeterministic net are *stop* transitions (in this particular model there was no need to use *run* transitions, which are required only when the probabilistic state change of one component comprises several steps and is modeled through a sequence of run transitions followed by the firing of one stop transition).

All places have color domain *Mach* so that the presence of a token of color $c_i \in Mach$ in a place represents the current state of machine c_i. For all transitions $compvar = x$, where x is the variable appearing on all arcs of both subnet.

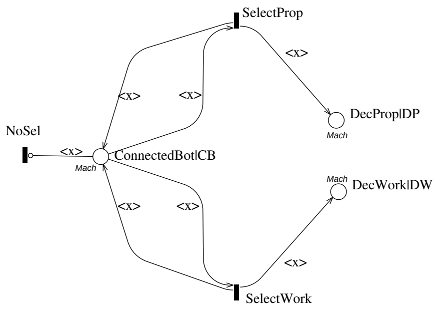
In Fig. 1.2(a) place *NoInfected* contains all the uninfected machines, while place *InitInf* contains all the infected machines that are not yet connected to the botnet. An uninfected machine can be infected by an active propagation bot (i.e., transition *BotInitInfection* where y represents such generic active propagation bot)[1] or by a

[1]Observe that y does not belong to the $compvar$ of transition *BotInitInfection*.



(a)

Transition	Weight	Prior.
NormalWork	6.0	4
InitInfectbyHum / BotInitInfection	0.4	4
BecomeActProp / BecomeActWork	0.15	1
BecomeInactProp	0.35	1
BecomeInactWork	0.15	1
RemInactProp / RemInactWor	0.8495	1
RemActProp	0.6	1
RemActWork	0.15	1
DetectActPropBot / DetectActWorkBot	0.05	1
DetectInactPropBot / DetectInactWorkBot	0.0005	1
BotConnection, BecomeProp, BecomeWork	1.0	2,3,3



(b)

Figure 1.2 MDWN probabilistic net with transition weights (a) and nondeterministic net (b) for the peer-to-peer botnet model.

hacker (i.e., transition *InitInfectbyHum*) during the initialization of the botnet. The inhibitor arc between place *InfectedMach* and transition *InitInfectbyHum* ensures that the hacker infection happens only when no infected machines are presented in the system (i.e., place *InfectedMach* is empty).

The firing of transition *BotConnection* moves a token from place *InitInf* to place *ConnectedBot*, and models the connection of an infected machine to the botnet. Subsequently, a role is associated with the new bot by the firing of transitions *BecomePropBot* or *BecomeWorkBot*: the former transition assigns a bot to the propagation activity, while the latter transition to the working activity.

A bot, whether assigned to the propagation activity or to the working activity, can either be inactive or active. Switching between these two states is modeled by the transitions *BecomeActProp* and *BecomeInacProp* (resp., *BecomeActWork* and *BecomeInacWork*). Finally, the recovery of a machine is modeled by transitions *DetectActPropBot*, *DetectInactPropBot*, *DetectActWorkBot* and *DetectInactWorkBot*.

All the probabilistic transitions have an integer priority used to avoid the confusion problem (i.e., transitions *NormalWork* and *BotInitInfection* have priority higher than those associated with transitions *BecomeActProp* and *BecomeInacProp*) and to reduce possible interleavings. The weight and priority associated with transitions is shown in Fig. 1.2(a) (bottom left corner).

The nondeterministic net N^{nd} in Fig. 1.2(b), models the nondeterministic choice to assign a role to a new bot. We recall from the introduction that the nondeterminism allows us to consider all the possible choices of malicious activities. Net N^{nd} shares the places *ConnectedBot*, *DecProp* and *DecWork* with the probabilistic net N^{pr}. For each bot in place *ConnectedBot* a nondeterministic choice is taken by the firing of transition *SelectProp* or transition *SelectWork*, which place a colored token in places *DecProp* and *DecWork* respectively. Transition *NoSel* is needed for technical reasons: it is used to conclude the nondeterministic step for all the machines not involved in the nondeterministic choice (i.e., not in place *ConnectedBot*). Also in this case priorities are used to reduce the possible interleavings.

All the MNDTSs in this model include $|Mach|$ transition instances, corresponding to a decision as to whether a just infected machine should be a propagating bot or a working bot (if a machine has not been infected yet, or was infected in some earlier step, no decision has to be taken and stop transition *NoSel* is included in the MNDTS for that machine). Also all MPTSs in this model include $|Mach|$ transition instances, each representing the probabilistic state change for each specific machine; observe that there are several self-loop transitions in the probabilistic subnet, representing the case when a given component remains in the same state.

Although this model has a single class of homogeneous components, one could easily refine it, e.g., to represent the fact that a subset $Mach_0$ of machines is protected by a firewall while another subset $Mach_1$ includes the machines in a DMZ. The model could then define different probabilities of infection for the machines in the DMZ and those protected by a firewall by properly defining $weight^{pr}$ as dependent on the color subclasses. It could also be interesting to add a non-controllable component representing a security software alternating between inactivity periods and security check periods (according to some probability distribution): during check

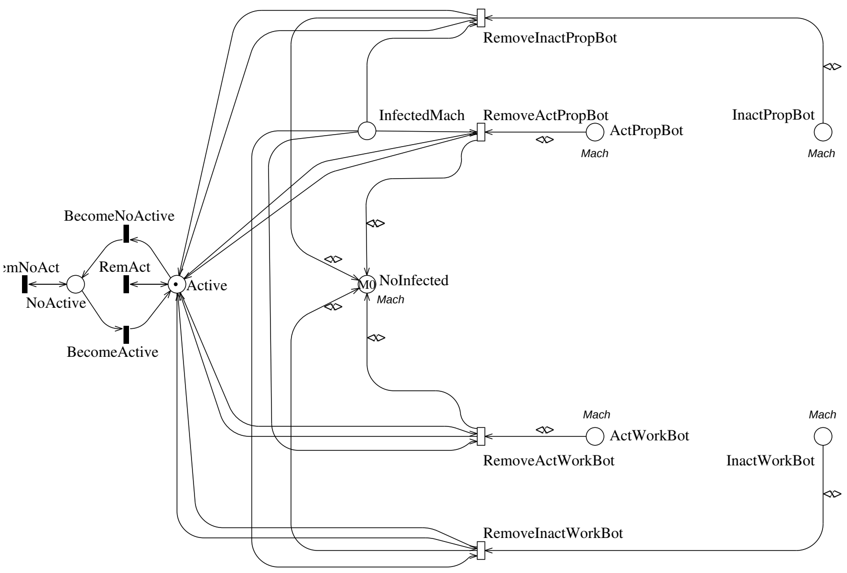


Figure 1.3 A sketch of the probabilistic subnet modeling security software

periods it synchronizes with the machine components to detect the malware and, in case it is found, removes it (a number of run transitions would be needed to represent the detection and removal of malware on all of the machines). Given that this component is non-controllable, it would not be involved as a component in any transition appearing in a MNDTS.

A sketch of the probabilistic subnet modeling this extension is shown in Fig. 1.3. where we have introduced two new places (*NoActive* and *Active*), encoding the security software status, and eight transitions. In detail, the stop transitions *RemNoAct*,*RemAct*,*BecomeNoActive* and *BecomeActive* are used to model the alternation between inactivity and security check periods. Instead, the run transitions *RemoveInactPropBot*, *RemoveActPropBot*, *RemoveInactWorkBot* and *RemoveActWorkBot* have a higher priority than all the other model transitions and represent the detection and removal of malware on all the infected machines.

An example of reward function for the botnet model could simply be a function providing the number of infected machines in each state (if, e.g., we want to study the strategy that maximizes the number of infected machines at time T, or the average number of working bots in steady state, or to compute the maximum probability that once the infection starts, all machines become infected within a given time T).

Computing and analyzing an optimal strategy using GreatSPN. Now we show how the described MDWN model can be analyzed studying the optimal strategy which maximizes a defined reward function. We used the tool MDWNsolver included in the GreatSPN framework for this purpose [6]. Among all the possible MDWN reward functions we choose the one that allows us to maximize the number of working bots, so rg is defined as: $rg = m(\mathit{ActWorkBot}) + m(\mathit{InactWorkBot})$.

The RG of this MDWN model with 10 machines has 3.42×10^{12} states. However efficient MDWN solution techniques [5], which automatically exploit the model symmetries to derive a lumped RG, can be used to reduce the size of the state space to 1.49×10^{7} . Then we generate the underlying (lumped) MDP, which has 7,722 states. All the process requires $\sim$30 min on an INTEL CORE I7 and $\sim$1GB of memory.

We recall that this difference in the number of states between the lumped RG of the MDWN and the MDP is due to the fact that the MDWN gives a component-based view of probabilistic and nondeterministic behaviors: complex nondeterministic and probabilistic behaviors are expressed as a composition of simpler nondeterministic or probabilistic steps, which are reduced to a single step in the final MDP.

The optimal repair policy is not trivial even if for each bot the system can choose only between two actions (i.e., *SelectProp* and *SelectWork*), and requires 10 min and 300MB of RAM to reach a precision of 10^{-8} through the policy iteration. The resulting policy provides for each state the best choice: in general it is extremely difficult to characterize in an abstract way the set of states that lead to a given action. Hence in practice some efficient representation of the table of all states associated with the optimal action should be used to apply the policy.

In order to study this optimal repair strategy we have computed the average number of working and propagation bots at time T by solving in transient the Markov chain obtained from the underlying MDP by fixing the action to take in every state according to the computed optimal strategy. The curves plotted in Figure 1.4 show these two performance measures for T from 0 to 500. The dashed lines represent the same measures computed when the choice between working and propagation activity is instead equiprobable. From this figure we observe, as expected, that the average number of working bots associated with the optimal strategy is greater than those computed according to an equiprobable choice (e.g., when $T = 500$ it is approximately 1.42 times larger)

Moreover, the histogram and curves show in Figures 1.5(a) and 1.5(b) show the effect of increasing the probability for a propagating bot to be active on the average number of working bots in steady state and on the average number of propagation bots at time T, respectively. These measures are derived by solving the MDWN model for different weights associated with transitions *RemActProp* and *BecomeInactProp* (i.e., (0,4,0.55), (0.5,0.45), (0.6,0.35), (0.7,0.25), (0.8,0.15), and (0.85,0.05), where the first element of each pair refers to *RemActProp*, the second to *BecomeInactProp*; in the labels of the figures, we indicate only *RemActProp*). Then the Markov chains obtained from these underlying MDPs by fixing the action to take in every state according to the computed optimal strategies are solved in steady state and transient to compute the two measures. The interplay between the probabilistic choice of propagating bots to remain active or inactive and the corresponding computed optimal strategy is not so obvious. An increase in the activity has two opposite consequences: a faster spreading on the infection, but also a higher probability to be detected (which is observed in Figure 1.5(b), which shows the transient behavior of

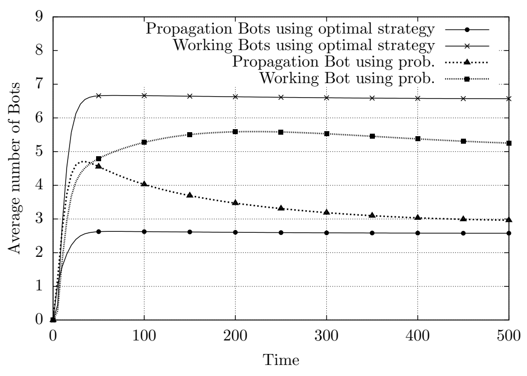


Figure 1.4 Average number of working and propagation bots using an optimal strategy or equiprobable choice ($0 \leq T \leq 500$, step 5)

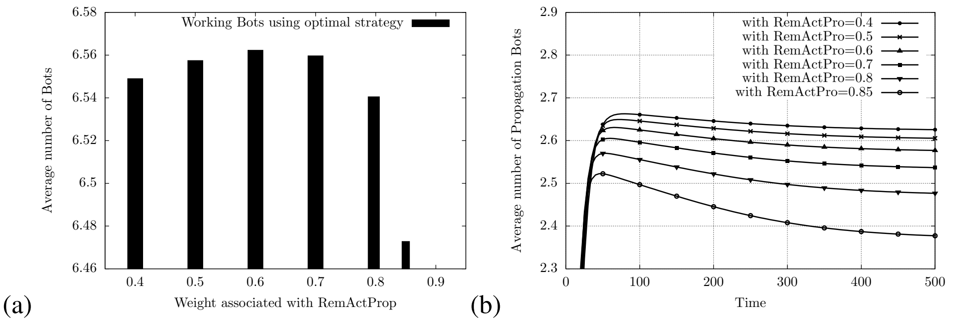
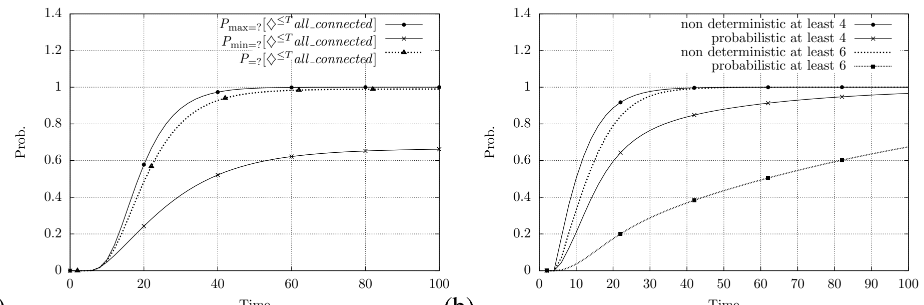


Figure 1.5 (a) Average number of working bots in steady state, and (b) average number of propagation bots at time T, varying weights associated with transitions *RemActProp* and *BecomeInactProp*

the number of propagation bots). In Figure 1.5(a), by increasing the weight associated with *RemActProp*, we observe that the effect on the reward (directly connected with the number of working bots) is first an increase and then a decrease.

Property verification using PRISM. We now consider analysis of the case study using the probabilistic model checking tool PRISM. In the analysis described previously, a single strategy satisfying an optimality criterion was generated *a priori* and then analyzed. In contrast, our subsequent analysis based on probabilistic model checking takes all strategies into account for each property considered. The properties will generally take the form of simple, non-nested properties of the probabilistic temporal logic PCTL [16, 8], although we also consider a property of PCTL*, in



(a) (b)

Figure 1.6 (a) Probability of all machines being connected to the botnet within time T, and (b) probability of at least 4 or 6 machines having the working activity within time T

which complex properties relating to execution paths can be represented, and a property of a reward-based extension of PCTL.

We used two methods to model the system in the PRISM input language. Firstly, we developed a translator to obtain a PRISM model from the lumped MDP generated by GreatSPN; this PRISM model comprises a single module with a variable for each place of the MDWN model. Secondly, we also developed a model of the botnet directly in the PRISM language, in which each machine is represented as a single module. Identical results were obtained for the same properties on the two models. The two models have different advantages and disadvantages: on the one hand, the model translated automatically from the MDWN results in a large input file, the parsing of which is demanding in terms of resources; on the other hand, the PRISM model with multiple modules is not convenient when modeling properties such as those concerning the total number of modules in certain states. Moreover, the symmetric update of all component state and the action choice for each component is obtained by explicitly modeling the "turns" in PRISM. In the case of the multiple-module model symmetry reduction [19], an operation which is similar to the generation of the lumped state space in GreatSPN, was used to reduce the size of the state space from 3.42×10^{12} to 5.3×10^{7}.

In Figure 1.6(a) we show how the probability of all of the machines being connected to the botnet changes with regard to the execution time of the system. The uppermost line of the graph represents the probability of the property written in PCTL-like syntax as $P_{\max=?}[\Diamond^{\leq T} all\_connected]$. The part $\Diamond^{\leq T} all\_connected$ of the property specifies that states labelled by the atomic proposition $all\_connected$ (which labels those states in which all 10 machines are connected) are reached within T execution steps. Instead the part $P_{\max=?}[\_]$ of the property specifies that the maximum probability of the inner part of the property should be computed, where the maximum is taken over all strategies. The lowest line represents the minimum probability of all of the machines being connected within a certain number of execution steps, written in PCTL-like syntax as $P_{\min=?}[\Diamond^{\leq T} all\_connected]$. The intermedi-

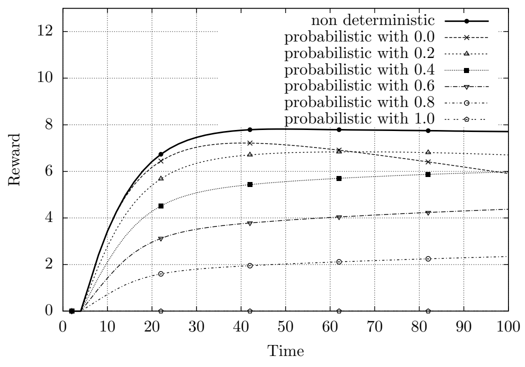


Figure 1.7 Average number of working bots at time T: nondeterministic model and probabilistic model, varying weights of selecting a machine for the propagation activity

ate line represents the probability of all machines being connected within a certain number of steps for a purely-probabilistic version of the model in which the choice between a newly-infected machine being made a working or a propagation bot is equiprobable.

We also consider properties concerning the number of working bots reaching a certain threshold. In this paper we consider the two thresholds of 4 working bots and 6 working bots, and the properties are written as $P_{\max=?}[\Diamond^{\leq T} \mathit{working\_at\_least\_4}]$ and $P_{\max=?}[\Diamond^{\leq T} \mathit{working\_at\_least\_6}]$, respectively. The results are shown in Figure 1.6(b). As in the case of the previous property, we observe a significant difference between the results obtained from our proposed model (denoted by "nondeterministic" in the figure) and those from a purely probabilistic model in which the choice between a newly-infected machine being made a working bot or a propagation bot is equiprobable (denoted by "probabilistic" in the figure). As an example of the resources used in one of the most costly verification instances that we considered, the result for $P_{\max=?}[\Diamond^{\leq 100} \mathit{working\_at\_least\_6}]$ required approximately 250MB and 84 min to obtain using PRISM's hybrid engine.

We also computed reward-based indices such as the maximum expected number of working bots at various time points (written in PRISM as $\mathbb{R}_{\max=?}[\mathtt{I}^{=T}]$); see figures 1.7. In this case, we were able to verify that, between 66 and 100 execution steps, the maximum expected number of working bots was greater than that for a number of different purely probabilistic models by a factor of at least 1.136 (approximately one machine), where the probabilistic models were obtained by varying the relative weight of choosing the propagation activity for a newly infected machine. This emphasizes the advantage of being able to change dynamically the policy of choosing between the activity of newly infected machines.

Finally, we also considered the maximum probability of the property that, after the initial infection, all machines become connected before the point in which all machines are removed from the botnet. We encode this property using a PCTL^* formula (for readers familiar with PCTL^*, the property comprises a number of nested until operators). The resulting probability was 0.9162633.

1.5 Conclusion

In this chapter the MDWN high-level formalism for the specification of MDP models has been introduced with the help of a case study on a peer-to-peer illegal botnet. After defining the MDP and MDWN formalisms, the properties that can be studied with different analysis methods are presented and illustrated on the case study: both transient and steady state behaviors can be of interest in the considered case study since it is relevant to observe both how quickly the infection spreads in the network, and the ability to survive of the botnet in the long run.

The aim of the proposed study is to show the range of possibilities that this kind of models provide for exploring the performance of alternative strategies, and to present some tools that can be used to this purpose. In particular two main approaches have been discussed: the former consists in solving an optimization problem, with the aim of maximizing a given reward function; in this case the goal is to maximize the steady state average reward. After having found an optimal strategy (at steady state), its transient behavior has been observed, and compared with that of alternative purely probabilistic strategies. This set of experiments has been performed with MDWNsolver, which is part of the GreatSPN suite. The other considered approach is based on the use of probabilistic temporal logic formulae to express properties of interest: in this case the study concentrated on a finite time horizon and used the PRISM tool. In this case it was possible to evaluate the minimum and maximum probability of the paths characterized by a given temporal logic formula, and the maximum expected reward at time T. Also in this case the influence of some system parameters on the maximum achievable value for the index of interest was shown.

The PRISM model used for the analysis of these properties has been automatically generated from the MDWN model using an ad-hoc translator (which needs some optimization in terms of the used exchange format). Another PRISM model has been built directly to compare the modeling convenience of the MDWN formalism and of the PRISM modular language, as well as to compare the effectiveness of the automatic state space reduction technique. The computation time and memory space requirements of the proposed methods have also been reported: it appears that increasing the size of the model (number of machines) above ten would require much more powerful computational resources.

In conclusion, the introduction of nondeterminism in stochastic models, typical of MDPs, allows one to explore behaviors that are not easy to observe in purely probabilistic models like Markov Chains; moreover the availability of high level formalisms for specifying MDPs can ease the task of modeling and analyzing complex distributed systems. The support of efficient tools implementing up to date research results is crucial for analyzing models of realistic systems.

Acknowledgments

The work of Giuliana Franceschinis has been partially supported by the GINSENG Project, Compagnia di San Paolo.

APPENDIX A

WELL-FORMED NET FORMALISM

In this appendix the formal definition and notation of the WN formalism is introduced.

WNs belong to the Colored Petri Nets (CPN) family; however the WN syntax has been carefully designed to allow the application of efficient analysis algorithms.

An WN N is a tuple

$$(P, T, \mathcal{C}, cd, I, O, H, \phi, prio, \mathbf{m}_0)$$

where:

- $P = \{p_i\}$ is a finite and non empty set of *places*.
- $T = \{t_i\}$ is a finite and non empty set of *transitions*.
- $\mathcal{C} = \{C_1, \ldots, C_n\}$ is the set of the *basic color classes*. A basic color class C_i can be partitioned into *static subclasses*: the number of elements of the partition of C_i is denoted $||C_i||$ and the j-th static subclass of C_i is denoted $C_{i,j}$, so that $C_i = \bigcup_{j=1}^{||C_i||} C_{i,j}$. The cardinality of a static subclass is denoted $|C_{i,j}|$. A color class C_i may be cyclically ordered.

- $cd(s)$ is a function which assigns to each $s \in P \cup T$ a *color domain* such that $cd(s) = C_1^{m_1} \times C_2^{m_2} \times \ldots \times C_n^{m_n}$, where the superscript $m_i \in \mathbb{N}$ depends on s and denotes the number of occurrences of C_i in $cd(s)$.
- A *guard* is associated to each transition t, and is a function $\phi : cd(t) \rightarrow \{TRUE, FALSE\}$ that is used to possibly restrict the color domain of t to colors satisfying a given Boolean condition. Each transition guard can be the constant function $TRUE$ or can be specified using a Boolean expression, whose terms are the *basic predicates*, described below.
- I (respectively O, H) assigns to each pair $(t, p) \in T \times P$ a function $I(t,p) : cd(t) \rightarrow Bag[cd(p)]$ (respectively $O(t,p), H(t,p) : cd(t) \rightarrow Bag[cd(p)]$); that is, $I(t,p)$, $O(t,p)$ and $H(t,p)$ map each color of transition t into a multiset of tokens on place p; $O(t,p)(c)$ will denote the application of the function to color c of t and $O(t,p)(c)(c')$, with $c' \in cd(p)$, will denote the multiplicity of color c' in the resulting multiset. WN's syntax for $I(t,p)$,$O(t,p)$ and $H(t,p)$ is defined below.
- $prio : T \rightarrow \mathbb{N}$ is the priority function defining a total order on the set of transitions.

A.0.1 Syntax of basic predicates

The WN syntax to express basic predicates (Boolean functions defined on $cd(t)$), needed to define transition guards, is defined as follows:

- $X_i^j = X_i^k$ (respectively $X_i^j \neq X_i^k$) is true if the elements of the argument corresponding to the j-th and k-th occurrence of class C_i are equal (not equal);
- $d(X_i^j) = d(X_i^k)$ (respectively $d(X_i^j) \neq d(X_i^k)$) is true if the elements of the argument corresponding to the j-th and k-th occurrence of class C_i belong (do not belong) to the same static subclass;
- $d(X_i^j) = C_{i,q}$ (respectively $d(X_i^j) \neq C_{i,q}$) is true if the element of the argument corresponding to the j-th occurrence of class C_i belongs to (does not belong to) static subclass $C_{i,q}$.

A.0.1.1 Arc function syntax.

The WN syntax to express functions labeling the arcs $I(t,p)$ $\big(O(t,p), H(t,p)\big)$ is defined as follows:

$$I(t,p) = \sum_i \lambda_i . F_i \circ [g_i], \quad \lambda_i \in \mathbb{N}$$

where:

- $[g_i] : cd(t) \rightarrow Bag[cd(t)]$ is a (guard-like) tuple selection function;
- $F_i : Bag[cd(t)] \rightarrow Bag[cd(p)]$ is a tuple of *class functions*;

- $\lambda_i \in \mathbb{N}$.

A *tuple of functions* F is such that

$$F = \langle f_1^1, \ldots, f_1^{m_1}, \ldots, f_k^1, \ldots, f_k^{m_k} \rangle$$

is defined as the Cartesian product of *class functions* $\{f_i^j\}$ where $f_i^j : Bag[cd(t)] \rightarrow C_i$. The application of tuple F to color $c \in cd(t)$ is defined as $F(c) = \otimes_{i,j} f_i^j(c)$ where $\otimes$ is the multiset Cartesian product operator.

The generic function f_i^j of tuple F is in turn a linear combination of *elementary* class functions, formally:

$$f_i^j = \sum_{k=1}^{m_i} \alpha_{i,k} X_i^k + \sum_{q=1}^{||C_i||} \beta_{i,q} S_{i,q} + \sum_{k=1}^{M_i} \gamma_{i,k} !X_i^k$$

where $\alpha_{i,k}, \beta_{i,k}, \gamma_{i,k} \in \mathbb{Z}$ and $X_i^k, !X_i^k, S, S_{i,q}$ are the elementary functions. Obviously, they have the same domain and codomain as f_i^j. More precisely:

- X_i^k, the projection: maps a color $c \in cd(t)$ onto the element of the argument corresponding to the kth occurrence of class C_i;
- $!X_i^k$, the successor: maps a color $c \in cd(t)$ onto the successor of the element of the argument corresponding to the $k - th$ occurrence of class C_i; it may be used only if class C_i is ordered;
- S_i, the constant function: maps any color $c \in cd(t)$ onto the (constant) multiset $\sum_{c \in C_i} c$;
- $S_{i,q}$, the constant on the q-th static subclass of C_i: maps any color $c \in cd(t)$ onto the (constant) multiset $\sum_{c \in C_{i,q}} c$.

A class-functions f_i^j must always map into multisets with non negative coefficients, so $\alpha_{i,k}, \beta_{i,k}, \gamma_{i,k}$ must be defined consistently with such assumption.

The function $[g_i] : cd(t) \rightarrow Bag[cd(t)]$ is right composed to tuple F and acts as the identity function for those elements of the domain which satisfy a given Boolean condition, otherwise it maps into the empty multiset. The Boolean condition is expressed using the same syntax defined for transition guards.

Remark: often in practice a more readable notation is used in annotations. In particular the name of color classes are chosen to suggest their meaning in the model, and the projection basic function X_i^j is denoted using a different symbol that corresponds to the name of a variable in the informal definition given earlier in this chapter. The variable names can be interpreted as "aliases" for the projection basic functions, which are more meaningful to the modeler.

A.0.2 Markings and enabling

A *marking* of place $p \in P$ is a multiset on the color domain $cd(p)$. $\mathbf{m}_0$ is the initial marking of the net and assigns to each place its local marking $\mathbf{m}_0(p) \in Bag[cd(p)]$.

An instance of t corresponding to color $c \in cd(t)$ is denoted $\langle t, c\rangle$. Instance $\langle t, c\rangle$ is enabled in marking $\mathbf{m}$ if $I(t,p)(c)(c') \leq \mathbf{m}(p)(c') < H(t,p)(c)(c')\ \forall p \in P, c' \in cd(p)$. An enabled transition instance $\langle t, c\rangle$ can fire, leading to a new state $\mathbf{m}'$ defined as follows: $\mathbf{m}'(p) = \mathbf{m}(p) - I(t,p)(c)(c') + O(t,p)(c)(c'), \forall p \in P$.

The constraints on the syntax of WN allow the automatic exploitation of the behavioral symmetries of the model and the possibility to perform the state-space based analysis on a more compact RG: the symbolic reachability graph (SRG). The SRG construction relies on the *symbolic marking* concept, namely a compact representation for a set of equivalent ordinary markings. A symbolic marking is a symbolic representation, where the actual color of tokens is forgotten and only their distributions among places are stored. Tokens with the same distribution and belonging to the same static subclass are grouped into a so-called dynamic subclass. Starting from an initial symbolic marking, the SRG can be constructed automatically using a symbolic firing rule [10]. Most qualitative properties of the model can be analyzed on the SRG. Moreover in [4] it is shown that in the context of MDWNs a lumped MDP can be directly derived from the SRG, allowing to compute more efficiently the same optimal strategy that might be computed on the ordinary RG.

REFERENCES

1. R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
2. C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.
3. C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.
4. M. Beccuti. *Modeling and analysis of probabilistic systems. Formalisms and efficient algorithms*. PhD thesis, Univ. degli Studi di Torino, Torino, Italia, 2008.
5. M. Beccuti, G. Franceschinis, and S. Haddad. Markov Decision Petri Net and Markov Decision Well-Formed Net formalisms. In *Proc. 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN'07)*, volume 4546 of *LNCS*, pages 43–62. Springer, 2007.
6. M. Beccuti, G. Franceschinis, and S. Haddad. MDWNsolver: A framework to design and solve Markov decision Petri nets. *International Journal of Performability Engineering*, 7(5):417–428, 2011.
7. R.E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.
8. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. 15th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.

9. H. Bohnenkamp, P. R. D'Argenio, H. Hermanns, and J.-P. Katoen. Modest: A compositional modeling formalism for hard and softly timed systems. *IEEE Transactions on Software Engineering*, 32(10):812–830, 2006.
10. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, 1993.
11. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
12. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.
13. L. de Alfaro. Stochastic Transition Systems. In *Proc. of 9th International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 423–438. Springer, 1998.
14. T. Dean and M. P. Wellman. *Planning and Control*. Morgan Kaufmann Pub., San Francisco, California, USA, 1991.
15. V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.
16. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
17. H. Howard. *Dynamic Programming and Markov Process*. MIT Press, 1960.
18. B. Jonsson, K. G. Larsen, and W. Yi. Probabilistic extensions of process algebras. In *Handbook of Process Algebra*, pages 685–710. Elsevier, 2001.
19. M. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In *Proc. 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4114 of *LNCS*, pages 234–248. Springer, 2006.
20. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
21. M.L. Littman, T.L. Dean, and L. Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proc. of 11th Conference on Uncertainty in Artificial Intelligence (UAI'95)*, pages 394–402, 1995.
22. A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Computer Society, 1977.
23. E. Van Ruitenbeek and W. H. Sanders. Modeling peer-to-peer botnets. In *Proc. 5th International Conference on Quantitative Evaluation of Systems (QEST'08)*, pages 307–316. IEEE Computer Society, 2008.