uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Luc Beaudoin

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (System Science)

GRADE / DEGREE

School of Engineering-System Science

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Autonomic Computer Network Defence Using Risk States and Reinforcement Learning

TITRE DE LA THÈSE / TITLE OF THESIS

Stan Matwin

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Nathalie Japkowicz

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Philippe Cabbé

Liam Peyton

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Autonomic Computer Network Defence

Using Risk States and Reinforcement Learning

Luc Beaudoin

Thesis submitted to
the faculty of graduate and postdoctoral studies
in partial fulfillment of the requirements
for the degree of Master of System Sciences (MSc)

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-61274-3
Our file Notre référence
ISBN: 978-0-494-61274-3

Table of Contents

List of Tables

List of Figures

List of Acronyms

CFNOC	Canadian Forces Network Operations Centre
CIO	Chief Information Officer
CRL	Collaborative Reinforcement Learning
CSEC	Communications Security Establishment Canada
CND	Computer Network Defence
CDR	Confidence Based Dual Reinforcement
DEDS	Discrete Event Dynamic System
DNS	Domain Name Server
DP	Dynamic Programming
DP	Dynamic Programming
FASM	Functional Application Support Manager
GPI	Generalized Policy Iteration
IT	information technology
LCAM	Life-Cycle Application Manager
LCMM	Life-Cycle Material Manager
MDP	Markov Decision Process
MTBR	Mean Time Between Failure
MTTR	Mean Time To Repair
MC	Monte Carlo
NIST	National Institute of Standards and Technology
NSR	National Service Releases
NVD	National Vulnerability Database
NOC	Network Operations Centre
NN	Neural Network
NATO	North Atlantic Treaty Organization
RL	Reinforcement Learning
SOX	Sarbanes and Oxley
SLA	Service Level Agreement
SA	Situational Awareness
TD	Temporal-Difference
TRA	Threat and Risk Assessment
US	User Service
WWW	World Wide Web

Acknowledgements

I would like to sincerely thank Dr. Stan Matwin and Dr. Nathalie Japkowicz from the University of Ottawa, for their scientific guidance throughout this project.

I would also like to express special thanks to Dr. Peter Mason and Dr. Julie Lefebvre from Defence Research and Development Canada, for their continuous support, mentorship and encouragement.

Above all, I wish to express my gratitude and love to my wife, Andrea, my sons Nicolas and Samuel, and my daughter Mia, who made countless sacrifices to give me the opportunity to complete this work.

Abstract

Autonomic Computer Network Defence aims to achieve self-protection capability of IT networks in order to limit the risk caused by malicious and accidental events. To achieve this, networks require an automated controller with a policy, which allows selecting the most appropriate action in any undesired network state. Due to the complexity and constant evolution of the Computer Network Defence environment, a-priori design of an automated controller is not effective. A solution for generating and continuously improving decision policies is needed.

A significant technical challenge in Autonomic Computer Network Defence is finding a strategy to efficiently generate, trial and compare different policies and retain the best performing one. To address this challenge, we use Reinforcement Learning to explore Computer Network Defence action and state spaces and to learn which policy optimally reduces risk. A simulated Computer Network Defence environment is implemented using Discrete Event Dynamic System simulation and a novel graph model. A network asset value assessment technique and a dynamic risk assessment algorithm are also implemented to provide evaluation metrics. This environment is used to train two Reinforcement Learning agents, one with a table policy and the other with a neural network policy. The resulting policies are then compared to three other empirical policies for their risk performances. These empirical policies serve as evaluation baseline and include: letting risk grow without any action, randomly selecting valid actions, and selecting the next action based on the pre-computed asset value of the affected assets and choosing the one with the highest value first.

We found that in all test scenarios, both Reinforcement Learning policies evaluated improved the overall risk when compared to random selection of valid actions. In one simple scenario, both Reinforcement Learning policies converged to the same optimum policy with better risk performance than other assessed policies. Generally, we found that, for the tested scenarios and training strategies, a simple policy addressing affected assets in the order of their asset values, can generally yield superior results.

Chapter 1

Introduction

1.1 Overview

Computer Network Defence (CND) is concerned with the active protection of information technology (IT) infrastructure against malicious and accidental incidents. Given the growing complexity of IT systems and the speed at which automated attacks can be launched, implementing timely and efficient network incident mitigating actions, whether proactive or reactive, is a great challenge [17,24,45]. A human is simply not able to handle the combination of the complexity and speed with which the analysis and response must be performed [8,9,14]. Therefore, some believe that in order to efficiently protect critical IT networks, CND actions selection and implementation should be automated [14,37]. We refer to this new area of research as *Autonomic CND*.

A system capable of achieving autonomic CND must be able to iterate through the CND decision cycle in an automated manner. This cycle typically involves the following steps: sensing network changes, analysing their impact, selecting an appropriate mitigation action, and implementing this action back onto the network. This process forms a control loop which employs available resources to continuously protect the IT infrastructure. Various commercial products and research prototypes support individual steps of this control loop. However, the search for an adaptive controller design capable of steering IT networks towards an acceptable and stable equilibrium in the face of security incidents is in its infancy.

A benefit of Autonomic CND is the potential reductions in the *risk* that network-enabled organisations may be exposed to over time. This benefit is achieved, at least conceptually, through automation of mitigating actions, or a subset thereof, which in turn, significantly decreases incident response time. This automation simultaneously frees existing resources for other non-automated mitigating tasks. Combined, the decrease in response time and increase in resources availability reduce the overall organisational risk exposure. In today's CND environment, where network security incidents results in billions of dollars in yearly loss to industries and governments, and where national' sovereignties are threatened by what is known as cyberwars and iwars [65], Autonomic CND offers much potential in better protecting critical IT infrastructures and reducing its associated costs.

In this work, we suggest, as our hypothesis, that autonomic CND can be achieved by using Reinforcement Learning, and Dynamic Risk Assessment to dynamically learn the optimal action sequence, or policy, to recover from given computer network risk situations. Such a policy could then be used by commercial network management and security products to implement selected mitigating actions automatically, as risk states are sensed.

The rest of this Chapter is organised in the following way: Section 1.2 discusses the problem of Autonomic CND and the motivation for this research and its hypothesis. Section 1.3 presents the contribution we made to this problem area and further expands on the organisation of this thesis.

1.2 Motivation

To understand our motivation for investigating Autonomic CND through the use of Reinforcement Learning and dynamic risk assessment, we first need to consider how risk influences CND action decisions today. Then, we need to consider how Reinforcement Learning, integrated with dynamic risk assessment, can successfully achieve automation of the CND decision process.

1.2.1 Risk in CND Decision Making: The Proactive-Reactive Dilemma

Prior to looking into the automation of CND, we should first describe some of the challenges associated with CND decision making today. We will first introduce a definition for CND itself and discuss typical CND actions and their triggering metrics.

The North Atlantic Treaty Organization (NATO) has defined Computer Network Defence as:

> *"Actions taken through the use of computer networks to protect, monitor, analyze, detect and respond to unauthorized activity within information systems and computer networks"* [1]

There are a number of research initiatives currently investigating areas identified by this definition, including the automated fusion of CND information for monitoring, detection and analysis purposes [4,21,50]. Although these remain active research areas today, ultimately, the value of performing CND actually comes from protecting networks and responding to events. If we focus specifically on the "protect" and "respond" terms of the CND definition, performing a CND action can be synonymous to implementing a ***reconfiguration action*** in order to move the network from an unacceptable to an acceptable state [37, 57, 76]. The decision to implement

[1] NATO publication 3000 TI-3/TT-1162

these actions can be taken either proactively (to protect), in anticipation of an event, or reactively (to respond), after the occurrence of an event. Reactive actions are usually triggered by the detection of *reachability* problems (ex: up-down status of an asset, or degradation of a link, reported using ICMP *PING* or by a user) and security alerts (ex: virus scanners and intrusion detection systems alerts). Proactive actions can be triggered by the disclosure of new vulnerabilities exposing IT assets to potential compromises, or by forecasting system requirements (system growth), also known as capacity engineering.

An important difference between proactive and reactive actions is the notion of *risk*. A proactive action aims to reduce the probability p of occurrence of a damaging event ($p<1$), whereas a reactive action typically aims to reduce the damage of an actual event ($p=1$). Often, both proactive and reactive actions rely on the same limited resources to be implemented. This leads to situations of conflicting priorities, where resources must be rationed between proactive and reactive tasks. An example of such *proactive-reactive* dilemma would be Network Operations Centre Staff (NOC Staff) needing to deciding between first patching a vulnerable system, or fixing an outage on another system.

To solve the *proactive-reactive* dilemma, we need a single metric which can account for both potential ($p<1$) and actual ($p=1$) incidents. This problem is also known as *decision making under uncertainty* and the metric we are describing is similar to *utility*. The notion of *risk* in Decision Theory is intertwined with *utility*. Most often, risk is defined as a weighted function of possible gain and losses with respect to their likelihood (*expected value*), although such risk can be skewed according to one's preferences (or risk aversion), in which case, a utility function must also be considered (*expected utility*). Presented as a problem of *decision under uncertainty*, it is foreseeable that the proactive-reactive dilemma can use *risk* as a threshold to choose between valid CND mitigating actions.

This approach however tends not to be the way CND decisions are made today. This is largely due to the complexity and uncertainty associated with *manually* computing this *risk* metric. It is important to realize that the rate of occurrence of potentially damaging events on today's IT infrastructures is far greater than the rate at which human resources can analyse them [4, 34, 51, 61]. As a result, the distribution of CND resources today is often skewed in favour of reactive ($p=1$) actions, regardless of the actual risk posed by potential damages ($p<1$). This bias, also known as risk aversion in Decision Making Theory is further reinforced by frequent self-inflicted collateral damages [78], which favours the decision to avoid taking proactive actions when not necessary.

As an illustration of this decision under uncertainty problem, consider the case of the internet worm Slammer. Slammer is an automated attack malware which infected over 75,000 hosts in 10 minutes, and caused massive disruption of critical IT services nation-wide in 2003. The outbreak

was eventually controlled by *patching* most vulnerable hosts (a massive vaccination if you wish). But the outbreak was avoidable since a patch was available six months before [34,51]. The proactive-reactive dilemma in this case consisted in balancing the resource cost associated with deploying the patch (also known as performing the patch management process, or the vulnerability management process), and the potential disruptions later realized by the worm outbreak. The latter being difficult to assess *a priori*, the decision made by many was not to proactively implement the patch.

Part of the problem is how to dynamically compute this *risk* metric as network conditions change. This is an open problem in network security as far as we know. It is composed of two distinct sub-problems: calculating the damages associated with an event, and assessing the probability of occurrence of a damaging event. Both of these problem areas are investigated in this thesis. Ultimately, this dynamic *risk* metric should serve as a feedback signal to steer the actions in the autonomic CND control loop.

It should be noted that in our work, we do not consider common existing risk assessment methodologies because they do not account for the dynamic nature of the CND environment. Although Threat and Risk Assessment (TRA) and security audits methodologies are quite popular today due to compliance regulations (including the Sarbanes and Oxley (SOX) Act, Canada's Bill 198, ISO 27002 and industry led PCI DSS), these do not account for response resources, nor enable dynamic re-assessment of risk once a network suffers damages. These properties are essential in the context of cyber warfare and automation. We identify these methods here because of the role they often play in today's IT infrastructure initial (*a priori*) defences' design.

1.2.2 Reinforcement Learning and Dynamic Risk Assessment for Autonomic CND

As we illustrated through the proactive-reactive dilemma, Dynamic Risk Assessment could greatly contribute to the optimal use of CND resources. However, it is insufficient to enable autonomic CND in itself. The issue is largely associated with the combined effects of time, security events arrivals and the selected actions on risk. Namely, one should consider factors such as:

- The expected time required to complete the mitigating action;
- The expected time before the next event arrival;
- The expected damages incurred if no mitigating action is taken;
- The expected business needs.

These factors lead to a combinatorial problem. Assessing the risk for every network state, action and event timing is path-dependent. This means that the overall risk exposure from a given situation depends on the entire sequence of actions taken to recover from it. To illustrate this important concept, a decision tree for a simple scenario with three (3) assets, only one of which being exposed to the Internet, is shown in figure 1.1. The first decision is triggered by a new vulnerability on *asset 1,* which is exposed to external exploit sources. From the resulting decision tree, we show seven (7) potential risk outcomes, depending on the timing of the first exploit event, the duration of each action, and the mitigation strategy used.

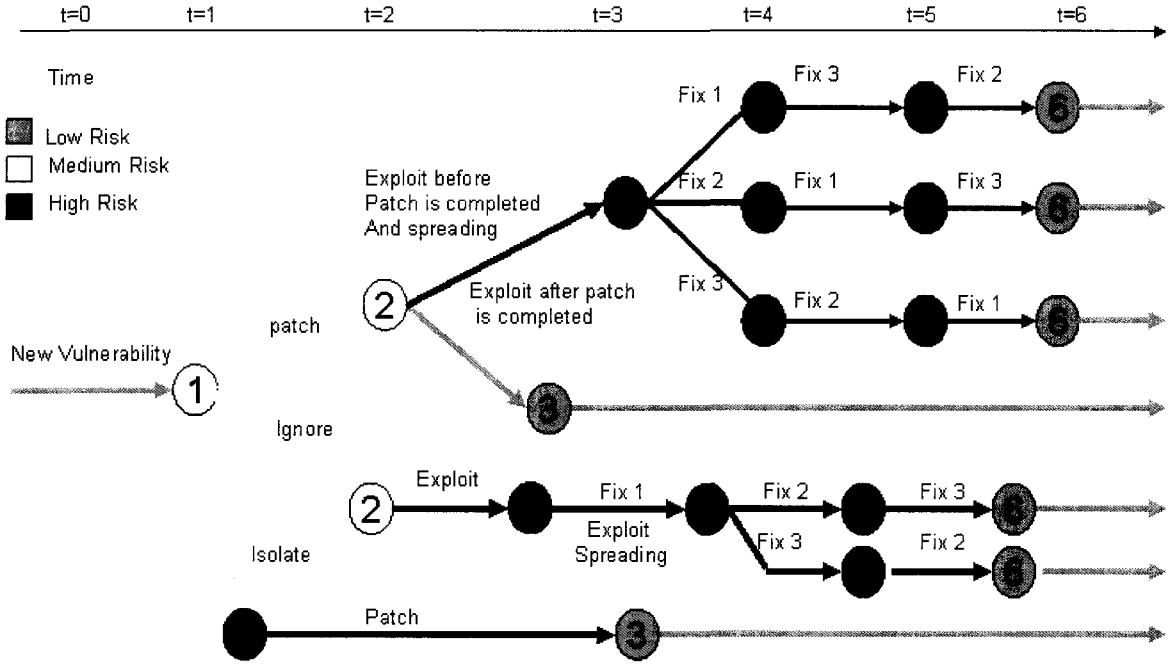

Figure 1.1 Example of a simple CND decision tree for a new vulnerability event

For this scenario, deciding to immediately patch *asset 1*'s vulnerability (node 1) may lead to an optimal path (green arrow starting at node 2, which means that low risk is assumed from this point on). However, the same decision may also lead to the worst path, in the case an exploit occurs before the patching action is completed, and spreads to the two other assets. For this reason, isolating the vulnerable host first, prior to patching it, may be an advantageous decision path. Although isolating *asset 1* results in self-denial of service, it also prevents any exploits from using this host to spread to other hosts. Once exploited, an asset has to be "fixed", which typically means restoring a clean disk image on the host.

In a continuous CND scenario, new vulnerabilities, exploits and outages can occur at any point in time, resulting in new decision branches for each arrival, many of which can be followed concurrently, or serviced through a queue, depending on the number of available resources. If we also consider that assets may have varying levels of business importance, the number of potential

outcomes quickly becomes unmanageable. This situation captures the problem we are interested in: given **vulnerability**, **exploit** and **outage** events, an IT network, stated business needs for IT services and limited resources, how can we decide which is the next action to take amongst options of **fixing**, **patching, isolating** risk-exposed assets, or simply **waiting**?

Assessing the cumulative risk for every possible branch, in a greedy way, is not a practical strategy to solve this problem. We need to be able to sample this state-action space and steer exploration towards the ***most promising action selection strategy***. This strategy is also known as a ***policy***, and the optimisation goal we seek is to ***minimize network risk*** over a given time horizon. The search for such an optimal policy is referred to as the Generalized Policy Iteration (GPI) problem, and different approaches have been used to solve it, including Monte Carlo (MC), Dynamic Programming (DP), and Reinforcement Learning. It can be shown that Monte Carlo and Dynamic Programming are closely related to (*special cases*) Temporal-Difference (TD) Reinforcement Learning [69]. Monte Carlo and Dynamic Programming largely suffer from "the curse of dimensionality", that is, the exponential explosion of the state space as state variables are added. Although this problem also applies to RL, many strategies exist to limit its effect, including eligibility traces and function approximation, which we will further describe in the next chapter. Furthermore, Monte Carlo and Dynamic Programming require that the transition function, that is the next state resulting from a given action, and its associated immediate *reward,* be known up-front. In a dynamic system like Autonomic CND, these parameters are not initially known and sampling of the various action sequences is required to know the resulting next state and risk benefit. Solving these situations of limited initial knowledge is the main reason RL was developed. As well, Reinforcement Learning has been successfully used in complex control loop systems, namely in automated packet routing problems [12][43][47] and automated server resources allocation problems [73]. Another benefit of Reinforcement Learning is that it can be used online (adaptation to situations as they occur), offline (planning for anticipated situations), on-policy and off-policy. The two last terms refer to whether the learned policy is used to influence exploration (on-policy), or whether the policy iteration is controlled through another mechanism independently, such as random selection, human control, or heuristic rules (off-policy). In other word, RL can learn how to optimally control a system through observation of its behaviours, or from collecting experience while actually controlling it, or even from both simultaneously. The particularities of these approaches will be further discussed in follow on sections.

The flexibility of RL in solving GPI problems encouraged our decision to use it to investigate Autonomic CND and the search for associated optimal policies. The use of Dynamic Risk Assessment as an objective function, or reward, to steer RL policy improvement also seems like a reasonable assumption which the work herein described attempts to validate.

1.3 Contribution to the field

For our contribution, we developed a novel framework to study Autonomic CND and assess CND policies. First, we developed a control system architecture which integrates CND events and resources with Dynamic Risk Assessment and Reinforcement Learning. We developed a Discrete Event Dynamic System (DEDS) simulator for the CND environment, which we integrated into our control system architecture, to provide us with a flexible experimentation framework. DEDS models are widely used in research of complex *man-made* systems. Even though they are removed from reality, they provide the ability to simulate with some fidelity systems' behaviours of interest which would otherwise be too dangerous or costly to observe. In the case of experiments with learning agents such as is the case in this work, DEDS allows running experiments which would take (many) life-times to perform otherwise.

We also contribute a novel algorithm allowing Dynamic Risk Assessment for network states. This algorithm includes a novel graph model of CND assets and their relationships to business needs, as well as a novel algorithm to compute individual network asset value. We explain how this risk metric can be used as an objective function to steer different Reinforcement Learning agents and policy models. Namely, we integrate into our experimentation framework two Reinforcement Learning policy implementations: one using a table which maps network states to preferred mitigating actions, and one using function approximators, specifically neural networks, to provide this mapping. As a prerequisite to implementing these RL policies, we propose a state vector representation for the CND environment which can hopefully reflect the punctual risk states. We then propose five different CND scenarios to experiment with this system and measure its ability to explore, improve and compare CND policies. They are: fixing four concurrent outages, fixing concurrent outages on all assets, patching concurrent vulnerabilities on all assets, patching a vulnerability or fixing the resulting exploits, and fixing, patching, isolating or waiting over a continuous ten year period simulation. These scenarios are applied to a simple CND graph with eleven assets and their interdependencies.

Once these contributions are introduced, we assess empirically the results of our asset value algorithm on the simple eleven asset CND graph. We demonstrate the validity of the asset value result by capturing a *trail of evidences* during the algorithm execution. Our assessment of these results shows that the algorithm produced intuitive asset values. We move on to assess the behaviour of our Dynamic Risk Assessment algorithm and show how risk is affected by multiple vulnerabilities on a host and how a discontinuity between potential damage ($p<1$) and actual damage ($p=1$) emerges and is corrected by the algorithm. We then present results from several discrete event simulation runs we conducted, using different policies as controller (for action selection). We collected data samples throughout these simulations allowing us to compare the different policies' performances using statistical analysis.

Based on these experiments, we show that our framework for studying Autonomic CND and assessing CND policies is flexible and supports complex scenario-based simulations. We show that it is possible to learn an optimal CND policy using Reinforcement Learning and Dynamic Risk Assessment as an objective function. We demonstrate that CND policies can be improved through RL, both for online and offline scenarios. We also show that in a simple case, such a *trained* policy can be optimum. More generally, we show that for our scenarios and training strategies, RL policies lead to better overall risk results than a random-valid action selection policy, but that a simple heuristic policy strictly based on selecting assets in decreasing order of their pre-computed asset value yields superior results.

1.4 Thesis Outline

We have organized this thesis into six (6) chapters. In **Chapter 1,** we introduce the topic of Autonomic Computer Network Defence and our motivation to investigate this topic in conjunction with Reinforcement Learning and Dynamic Risk Assessment. In **Chapter 2,** we formally introduce Reinforcement Learning and review prior work pertinent to Autonomic CND, including autonomic computing, security policy management and network risk assessment. **Chapter 3** presents our Autonomic CND experimental framework architecture. Namely, we describe each framework module, the CND Discrete Event Dynamic System simulation developed, a novel Dynamic Risk Assessment algorithm, and the specific RL algorithms and policies integrated. In **Chapter 4,** we present a simple CND environment and test scenarios. **Chapter 5** presents our results for each test scenario simulation. We discuss these results in **Chapter 6**. Finally, we conclude with a summary of our findings, their limitations and proposed future work in **Chapter 7.**

Chapter 2

Related Works

We introduced in Chapter 1 our research objective to investigate Autonomic CND using Reinforcement Learning to explore and improve CND policies, steered using Dynamic Risk Assessment. Our motivation for choosing this approach was described along with some characteristics of the underlying technologies. We will now explain in greater details the field of Reinforcement Learning and the Generalized Policy Iteration problem, and how Discrete Event Dynamic System (DEDS) simulation is closely related to these domains. We will then explore the state of research in Autonomic CND. Although the field is in its infancy, it borrows from many existing research areas such as Autonomic Computing, Automated Policy Management and security metrics, for which we will summarize some of the associated research.

2.1 Reinforcement Learning

Reinforcement Learning is a family of machine learning algorithms which allows learning how to accomplish a task based on rewards and punishments. An excellent review of Reinforcement Learning is presented in [36] and [69]. RL evolved from the field of optimal control Dynamic Programming (DP), introduced by Bellman in 1957, Markov Decision Process (MDP), and trial and error behaviourist studies in biology and psychology. In essence, RL provides a way to solve what Minsky described in 1961 as the "credit assignment problem", that is: in various sequences of actions leading to a desired goal, how can we evaluate the individual contributions of each action, hence allowing finding the optimal sequence, or policy. In order to fulfill this learning task, RL makes use of the Law of Effect, first introduced by Thorndike in 1911, reinforcing events based on the tendency to select a given action.

Many variants of RL exist, but they essentially all attempt to solve the Generalized Policy Iteration problem. There are two concurrent tasks associated with policy iteration: finding the optimal objective function V^*, and learning the optimal policy π^*. As an example, in CND, the optimum objective function may be the minimum risk exposure achievable from an initial vulnerable situation, whereas the optimal policy may be the best patching sequence to achieve this minimum risk exposure. As the solution landscape is explored, RL allows *following* progress made on both the value function and the policy to find the point where both converge.

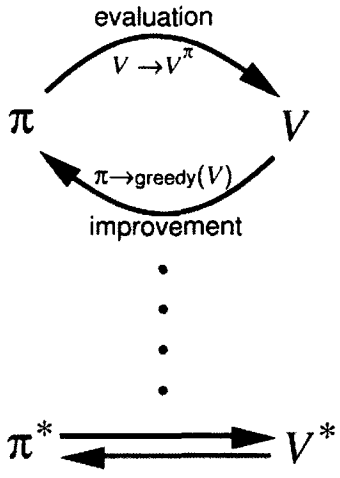

Figure 2.1 Generalized Policy Iteration[2].

To achieve this, the simplest form of RL focuses on state value updates following these steps: observe a state $\mathbf{s_t}$, select an action $\mathbf{a_t}$, observe the new state $\mathbf{s_{t+1}}$ and receive a reward $\mathbf{r_{t+1}}$, then update $\mathbf{s_t}$ value and repeat. The way RL updates the value of an action from the reward it receives follows the principle that actions performed closest the goal have more value. Upon reaching such goal, a reward scalar is received for the last action selected. The action's reward ***r*** is depleted by a factor gamma (γ) between 0 and 1, and used to update iteratively all preceding actions, in a backward propagation way. As a result, following a path of state-action, or policy π, from a given state $\mathbf{S_t}$ yields a cumulative reward value ***V***, or *return,* for this state according to:

$$V^{\pi}(S_{(t)}) = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

Finding this quantity for each state may be done recursively. For a given policy, the value of a state ***S*** is equivalent to the immediate reward received plus the accumulated value of the follow on state ***S'***, and so on, as follows:

$$V^{\pi}(s) = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

The optimal policy, $\boldsymbol{\pi^*}$, will optimise this value ***V**** for all state ***S***. This value function can also be extended to be probabilistic by summing over a density function. During a learning episode, RL must update its current estimate of ***V*** until it converges to ***V****. There are various strategies for this updating process. One may take the average of the rewards received over all the exploration steps, or use a bounded horizon, that is, an average over only a certain number of steps, to

[2] From Sutton: Reinforcement Learning, an Introduction, book on-line at http://www.cs.ualberta.ca/%7Esutton/book/ebook/node12.html

compute the discounted cumulative reward value V. The question of *when* and *how* to update a state value yielded many variants of RL algorithms. Rewarding earlier than later, as Mitchell puts it, is generally a reasonable assumption [54]. The notion of a learning rate, α, is also used to balance the weight of recent updates against previous estimates of V. The learning rate may be adjusted to be high at the beginning of exploration and be reduced after much experience is collected. Generically, an update function may look like:

$$V^{\pi}(s) \longleftarrow (1-\alpha)V^{\pi}(s) + \alpha(R(s,\pi(s),s') + \gamma V^{\pi}(s'))$$

The task then becomes finding the policy which will optimize V for all states S. If V is known, this can be done by following the maximum value of V received for the discounted follow-on state:

$$\pi^{optimal} = \arg\max_{a}[R(s,a) + \gamma V^{*}(\delta(s,a))]$$

Where R*(s,a)* is the immediate reward received, V^* is the optimal value function for $\delta(s,a)$ which is the next state after applying action a to current state S. This expression does not make use of π to evaluate the next state. Rather it generically uses the transition function $\delta(s,a)$. This allows sampling the unknown underlying decision process by trying actions according to empirical rules. This approach is referred to as off-policy exploration. When exploration relies on the existing estimated policy to select actions, it is referred to as on-policy. This notion splits RL algorithms into two families, Dynamic Programming on one side (off-policy), and Markov Decision Processes on the other (on-policy), with a number of RL hybrid methods in between. The simplest RL algorithm is called Q-Learning, and is an off-policy method essentially equivalent to dynamic programming. For a state estimate, Q-Learning replaces Bellman's conditional expectation by a value function called Q-Value, as shown below:

Q-Learning (Q-Function):	$Q(s,a) = r(s,a) + \gamma V^{*}(\delta(s,a))$
DP Bellman equation:	$E[R \mid s_t] = r_t + \gamma E[R \mid s_{t+1}]$

Q-Learning has the benefit of learning the optimal objective function (Q^* in this case) and the optimal policy concurrently. This is done by replacing the transition function by an *exploration policy*. Two main approaches exist: ε-greedy and Boltzman. These approaches aim at balancing exploration with exploitation. In ε-greedy, the state-action yielding the highest Q is selected with probability ε, otherwise, a random action is chosen. In Boltzman, a probability distribution for each action is derived from existing estimates according to:

$$P(a \mid s) = \frac{e^{\left(\frac{Q(s,a)}{\tau}\right)}}{\sum_{a'} e^{\left(\frac{Q(s,a')}{\tau}\right)}}$$, where τ is the "temperature" (analogy to simulated annealing)

Separating the exploration policy from the learned policy is fairly important. Otherwise, the initial policy has too much influence on the ability to converge to V^*, as is the case for pure on-policy methods. In the latter case, the initial policy needs to be fairly good initially.

Another important characteristic of RL is that the iterative process requires that each state be visited a number of times to converge to V^*. For DP, MDP and RL, this is provable only after an infinite number of iterations. In practice however, policy convergence occurs before value convergence. In other words, the relative ordering of state-action values, for the greedy case, remains unchanged between iterations, even though values keep being updated.

Since RL was introduced by Sutton, Barto and Watkins in the late 1980's [68], many strategies to accelerate policy convergence have been developed. One proposed approach is to update many state-action pairs each time a reward is received, rather than only updating the current state value estimate. This set of states to be updated is referred to as eligibility trace. Prioritized sweeping is another of these techniques which selects states to be updated using random sampling. The Temporal Difference (TD) learning algorithm [69] allows one to define the decay function of the reward over the "trace", essentially allowing changing the time horizon over which experience collected should be considered for each update.

Another convergence improvement approach was the introduction of hybrid algorithms capable of inductive and analytical learning. This is the case of DYNA [68], which updates both its estimation policy and behavior model (exploration policy) at the same time. This approach of using a model as part of the algorithm is also referred to as "planning".

A common way to illustrate how RL works is using a maze problem such as the one shown on the left side of figure 2.2. In this maze, the exploring *agent* must reach one of the goal states (marked as **G**). Let us assume that the initial value function V is set to zero for all states **s** (this is not always the case). During exploration of the maze, the agents selects randomly (pure off-policy) between four actions: up, down, left, and right, until the agent *luckily* finds a goal state. Initially, each transition between states leads to no reward. However, once a goal state is reached, the adjacent state value V is incremented by the goal-reward, discounted by gamma. Repeating this process sufficiently often eventually back-propagates the discounted goal-reward to all states. From any state, if we indicate the action which leads to the next state of highest value V, we obtain an optimal policy similar to the one presented on the right side of figure 2.2 (arrows show preferred actions for every state).

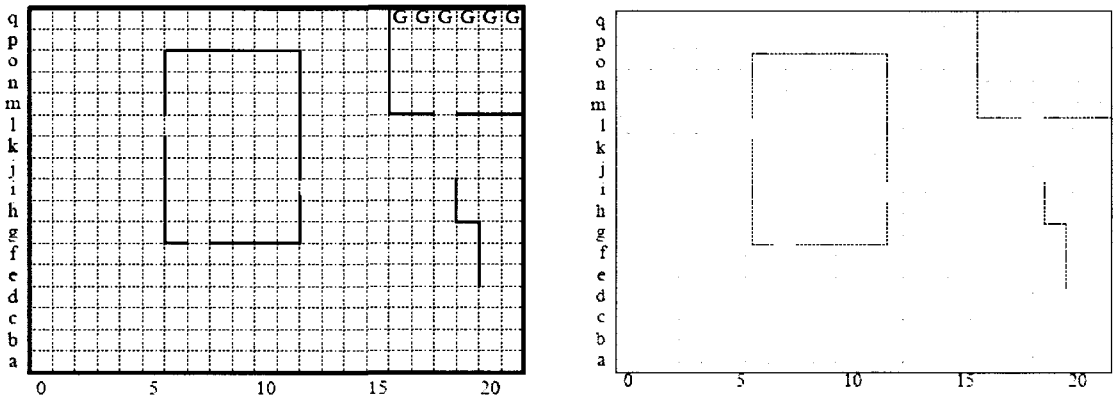

Figure 2.2 Reinforcement Learning policy obtained from a maze problem. [36]

A practical issue in RL is the storage of the policy. A common approach is using a table of state-action-value tuples. The maze on the right can be seen as the optimal policy, which results from a greedy search over the whole table-policy for actions with the maximum value for each state (shown by the arrows). The size of the whole table-policy is equal to:

$$PolicyTableSize = Size\_axis(x) * Size\_axis(y) * Actions(number)$$

The axis ***x,y*** and ***actions*** are referred to as *policy parameters*. Other applications may require more parameters, resulting in much larger policy spaces. Although memory and computing power today have permitted using RL with table policies in environment of several millions states, this approach remains unsuitable for many environments, including continuous state space problems. As a result, function approximators were merged with standards RL algorithms to *generalize* the state space. This approach described by Sutton in [70] proved to be successful in many applications with large and continuous state spaces such as robot control [48].

The idea of using function approximators in RL is aimed at avoiding storing every pair of *discretized* state-action-state-reward transitions. This can be accomplished by replacing the table policy by a function approximator using states as input and actions' value ***V,*** or ***Q,*** as output, as shown in figure 2.3. This potentially reduces the memory requirements, depending on the function approximator used.

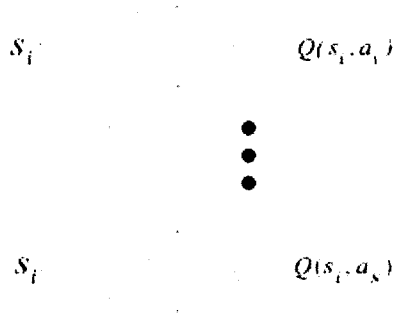

Figure 2.3 Function approximator for each action, with system states as input and Q-values as output [48].

In this example, back-propagation neural networks are used as function approximators. The RL algorithm requires slight modification to fit the neural-network learning paradigms. The reward received needs to be translated into an error minimisation metric for the neurons update algorithm. The idea is that the error will decrease through trial and error and the policy will converge to the optimal solution. The neuron error function, for the case of one-step Q-learning, is given by the following equation:

$$e_t = r_t + \gamma * \max_{a \in A} Q(x_{t+1}, a_{t+1}) - Q(x_t, a_t)$$

This error function represents the difference between the current value estimate and the updated value estimate by using the Q-learning update algorithm described earlier. The implementation of this algorithm is called Q-Connectionist [48]. Figure 2.4 shows a high-level architecture of the Q-Connectionist learning agent.

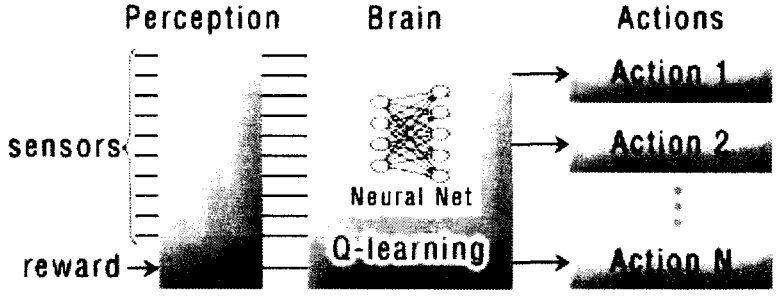

Figure 2.4 Q-Connectionist framework high-level architecture[3].

[3] Framework implementation available in Java source code at http://www.elsy.gdan.pl/

The Q-Connectionist approach can also use eligibility traces which are used to update different action-specific neural networks in the following manner:

For a trace with discrete points Pi , of size n +1 :

1. $i \leftarrow n$
2. $e_t \leftarrow Q(x_t, a_t)$
3. $u_{t+1} \leftarrow Max\{Q(x_{t+1}, k) / k \in A)\}$
4. $e'_t \leftarrow r_t + \gamma * [(1-\lambda) * u_{t+1} + \lambda * e_{t+1}]$
5. $UPDATE - NEURONS - NN[Q(x_t, a_t)] : error = e'_t - e$
6. $if(i = 0, quit), else(i \leftarrow n-1, goto\_step2)$

Table 2.1 Q-Connectionist eligibility traces update rule.

2.2 Reinforcement Learning, Discrete Event Dynamic Systems and Policy Iteration

The RL methods we have described attempt to learn an optimal policy by first finding its value function *V*. It is however possible to learn this policy directly without the value function. This can be accomplished using policy gradient, which essentially attempts to control policy parameters updates through hill-climbing towards highest reward gradients. This method is subject to getting stuck in local optimum but can be quite efficient when the initial policy is "good enough". An active field of research in policy parameter sensitivity is called Perturbation Analysis, and is used in Discrete Event Dynamic System (DEDS) modeling and simulation. DEDS was developed specifically for studying complex man-made systems' behaviors such as IT networks.

The integration of DEDS, Perturbation Analysis and RL is described by Cao in [16]. In Perturbation Analysis, differences of performance between two sets of policy parameters are defined as *potentials*, and comparing *potentials* yields a *realization factor*. These quantities have been shown to be related to RL *rewards*. Furthermore, Cao shows that DEDS perturbation analysis leads naturally to policy iteration. Cao also suggests that merging DEDS with RL can enable performing policy iteration and optimization more efficiently.

This approach of using DEDS with RL to explore policy space was also used by Dadone et al [20] to design (off-line) an optimal machine repair controller for a production plant, and to improve it on-line. In there system, Dadone used Mean Time To Repair (MTTR) and Mean Time Between Failure (MTBR) as reward function to learn the policy.

Another classic example of DEDS and RL is the improvement of an elevators dispatch algorithm proposed by Barto in [3]. The state space for the elevator dispatch problem was conservatively estimated at 10^{22} states. This work made use of DEDS with Poisson inter-arrival rate between elevator *consumers*. It merged continuous time events with discrete-time reinforcement learning. Barto suggested that state value function be defined as an integral rather than a discrete discounted sum, as follows:

$$V(s_t) = \int_0^\infty e^{-\beta\tau} r_\tau d\tau ,$$

where r_τ is the instantaneous reward and β controls decay, like γ does in the discrete case.

In this application of RL with DEDS, Barto shows that the policy learned is superior to common existing elevator dispatch algorithms, even though the branching factor is effectively infinite, due to the random continuous time of arrivals. The latter observation was acknowledged to put considerable strained on the RL agent, namely for the requirement to perform state look-ahead. Barto also varied the randomness of the exploration policy as the training progressed. Although the results were impressive, more than four days of simulation were required to train the agent.

These applications of DEDS with RL support our hypothesis that policy iteration in CND space may be accomplished by merging these two technologies.

2.3 Autonomic CND

As previously pointed out, Autonomic CND is a new domain of research, but is closely related to other self-managed system research areas. We introduced CND earlier as a field merging network security and network management. Automation research in these two areas is referred to **Autonomic Computing** and **Automated Security Policy Management**, respectively. In this section, we review some of the research in these respective areas, underlying their relationships with RL and DEDS simulation, as well as their use of different metrics.

2.3.1 Autonomic Computing

Autonomic Computing is a field a research focussed on self-managed computer systems. In their survey of the field, Dobson et al [24], supported by Haverkort [37], point out from the very beginning that *a priori* design of a control strategy for such systems is "useless" due to the

dynamics involved. They both clearly state that an adaptive controller is necessary. Some of the adaptive approaches investigated in their review include distributed algorithms such as morphogen-gradient approaches, which use the notion of a "field" signal, as a control metric, to steer agent-groups adaptation. They also identify game theory and reinforcement learning as promising techniques in the search for an appropriate adaptive controller for Autonomic Computing.

In another autonomic system survey, by Bertels et al [9], a distributed architecture of autonomic elements was controlled by Service Level Agreements (SLA). Such architecture is further defined by Garlan in [31] through the use of model-based adaptation for self-healing IT systems. Garlan's work focussed on repair actions which are taken from a predefined set of fixed policies. This approach, which was also taken by Haverkort in [37] to maintain quality of service through self-configuration, requires that a large number of stochastic models be pre-generated. Unlike reinforcement learning based policy, both Garlan and Haverkort's approaches rely on very accurate initial models and cannot adapt to unforeseen conditions. Another limitation of these approaches concerns the thresholds used to trigger actions, which were mainly based on bandwidth and servers' load. These metrics represent local measures and do not capture a holistic system state, which would be desirable for CND applications.

A more adaptive approach was proposed by Dowling in [26], called K-component model, which uses distributed agents exchanging asynchronous event messages to partially sense the environment and communicate between one another. Dowling proposed using Collaborative Reinforcement Learning (CRL) to solve this discrete event-driven optimization problem and share an optimal policy amongst agents. The K-component concept proposed with CRL was applied to the network load balancing problem. This work has both the benefit of presenting an adaptive and scalable approach as well as a holistic performance measure. Unfortunately, this research did not include experimental results, as it is often the case in this field.

Another autonomic computing approach was proposed by Tesauro, who is also credited for developing the reinforcement learning TD-Gammon system in 1992. In an IBM manifesto on RL and Autonomic Computing [72], Tesauro proposed solving the problem of optimal allocation of server resources to respond to client requests, considering service level agreement (SLA) contracts, using a reinforcement learning controller. Tesauro demonstrated the concept using server queues, average number of *http* requests per seconds and processors' load as input state metrics, to optimally decide whether to increase or decrease the number of servers allocated. In this work, Tesauro used discrete event simulation and a cost utility function, updated at a fixed five (5) seconds interval, to provide discrete rewards to the learning agent. This sampling is necessary when merging DEDS with continuously utility function, which is an important observation for our Autonomic CND prototype.

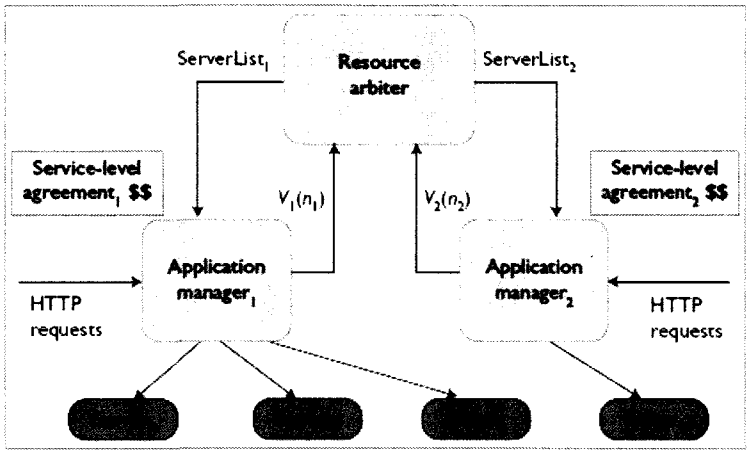

Figure 2.5 Reinforcement Learning in Autonomic Computing for server revenues optimization. [72]

Another popular autonomic system research area focuses on the optimal routing of network packets. Although the existing approaches, based on Bellman-Ford algorithm (Dynamic Programming) proved to scale to the current size of the internet, they are also limited in their ability to consider richer utility functions, including considering traffic types, or some special destination, as an example. For these reasons, Boyan and Littman [12] proposed an algorithm called Q-routing, which was used in a simulated experiment to replace common shortest path routing tables by Reinforcement Learning table policies, as shown in figure 2.6. The algorithm was later improved by Kumar [47], who introduced an algorithm called Confidence Based Dual Reinforcement (CDR) Q-Routing. Kumar added to Q-routing the notion of forward and backward exploration to update two states concurrently. Both Q-routing and CDRQ-Routing were demonstrated in discrete event simulations and both significantly reduced packet average delivery time.

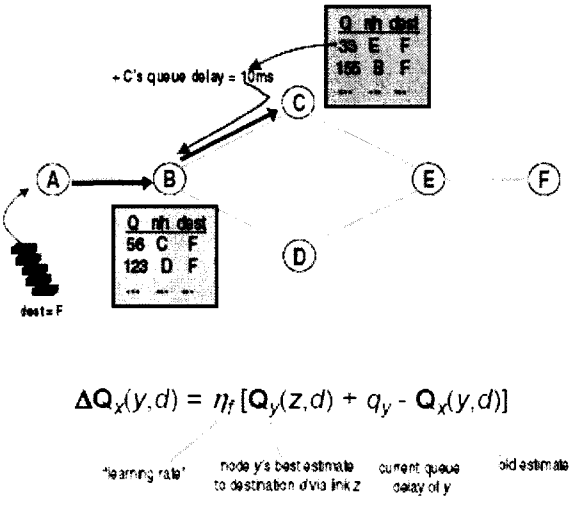

Figure 2.6 Replacing routing tables by Q-learning policy table in a routers network [12].

From our review of Autonomic Computing, we found no evidence of operational implementation of the concept. We found that most experimental results were derived from limited simulation environment and that the tested environments were of small size. We also found that RL is often used in Autonomic Computing research, but that the state space and policy sizes used are limited. These observations speak to the infancy of the field and the technical challenges involved. Adding security to Autonomic Computing is expected to further increase the domain complexity. Nevertheless, as an initial experimentation strategy for Autonomic CND, simulation paired with manageable environment state and policy sizes seems like a well supported approach.

2.3.2 Automated Security Policy Management

Efforts in the realm of network security automation has led to significant research on improving sensor events accuracy as well as analyzing security postures of given infrastructure. These efforts are captured in research on intrusion detection system alerts fusion [4], event log correlation [80], and model checking [63] [77]. In this section, we focus on the problem of automation of actual network reconfiguration action. Automated reconfiguration for security is an area of research very much in its infancy. Most of the related work falls under the theme of security policy management.

Automation of security policy management has been discussed by Burns et al. in [14]. Their work aimed to verify that security policies can dynamically uphold as the network changes. Burns put a large focus on finding a policy definition language to capture what is referred as *invariants*, or network "good states". They defined response actions mainly as homeostatic, that is, attempting to maintain a known good state. Burns et al. also proposed using a policy engine paired with sensors and policy enforcement elements to automate policy management in a control loop. The policy engine proposed was essentially a rule-based validation engine which mapped network configurations to previously defined "good states". A more adaptive model was proposed by Benjamin in [8] and used prior knowledge in a similar control loop to prune security actions from response policies. In this work, responses included *refresh, reset, ping, quarantine, isolate* and *degrade*. Benjamin proposed comparing its response selection method against random action selection and heuristics such as ordering responses based on cost and severity, although no mention was made on how these metrics were computed. In a final note, Benjamin identified the potential of using learning, including RL, for automating their security policy control loop.

A framework for adaptive security policy in a control loop was also proposed by Feiertag in [28]. The resulting system project, funded by DARPA, called UltraLog, was intended to allow

dynamic adaptation while providing strong assurance of consistency over system requirements and an intuitive interface to non-expert policy managers. Feiertag et al. described the need for both policy selection and generation, and acknowledged that such policies should account for available resources. One major challenge identified in their work is the ability to capture the intent, or objective, of security policies, and generate such a set of policies for evaluation.

Another approach to adaptive network security was proposed by Kotenko in [46] using multi-agent modelling. Kotenko defined three levels of cyber-security: *static*, which include "traditional"crypto-graphic, authentication and access control methods, *proactive* which includes network state monitoring, and *managed*, which includes optimal response selection and adaptation. Kotenko developed a discrete packet event simulation environment using OMNet ++ INET, which captured both defending and offending agent teams, and their respective strategies. The actions taken by the defending teams consisted in allowing or denying certain packets. The overall performance was assessed in terms of dropped legitimate packet rate (false-positive). Kotenko performed perturbation analysis of the resulting metrics against changes to the simulation input parameters, which included network configuration, attack intensity, defense mechanisms and defense teams' distribution.

A similar effort used game theory modeling to capture attacker-defender interactions in a simulated network environment. Bursztein [15] proposed using a goal called "Strategy Objectives". This strategy is analogous to what we formerly defined as a policy. This work used a model checking approach, derived from the Anticipation Game, to build a network security set of predicates, similar to what Ou in [60] and Wang in [77] have done for attack graphs. The model however used rewards and costs between the different actions taken, and also accounted for discrete time sequences, which is not typically found in other attack graph models. Interestingly, Bursztein used a well known MDP algorithm to find vertex values in the graph model, called PageRank (Google), which leverages vertices dependencies. The author could not interpret the results obtained by this approach, stating they were nevertheless "interesting". Bursztein's work concluded that the policies obtained were valid but not complete, that is, could not provably be global optimums.

A recurrent theme in security policy management is the notion of security metrics and policy goal. Researches in model checking mostly provide a binary policy metric by verifying compliance against requirements (pass or fail). One challenge in obtaining a more granular assessment of security metrics is prioritizing network assets by value. This issue was identified by Wang [77] and Moitra [55], and is generally recognized in attack graph and model checking research. Most of these models either use uniform assets values, manually inputted values based on SME's opinions, or non-intuitive MDP computed asset value such as Bursztein's method described earlier. During this thesis work, we published in [6] an asset valuation technique using

a user services model dependency graph. The benefit of this approach is that only services values are initially required; all other assets inherit their value from these services.

The asset value metric is critical in assessing a damage function for security. This is further supported by Moitra in [55] who also conducted analysis of survivability of network systems. In their work, Moitra et al. used discrete event simulation with incidents and exploits events arrival modeled according to Poisson distributions, to evaluate survivability of networks against various responses. The survivability metric used was computed by the following equation:

$$SURV(s) = \sum_{k} w(k) * \vartheta(s,k)$$

Where k is a service provided by the network, $w(k)$ is the service importance (asset value), and $\vartheta(s,k)$ is the degree to which the service k survives for network state s. Moitra did not go into further details as to how these quantities can be computed.

Another approach to measuring security metrics was proposed by Lefebvre et al in [50]. The model they developed, shown in figure 2.7, fuses network configuration (IT Infrastructure Resource Model), IT services value (operational capability requirements), exploits, vulnerabilities, safeguards, threats, alarms, and events into an information requirements model which produces *impact* and *risk* metrics. One of the sub-information products of Lefebvre's model, also referred to as the CND Situational Awareness (CND SA) model, is the list of critical resources assets, which is a product of operational requirements and IT infrastructure dependencies (label ***2*** in figure 2.7). It basically supports the idea that a network's goal is to offer services to meet organization's needs, which is similar to SLA objectives used by Tesauro's autonomic computing model. Although the CND SA model does not explicitly account for temporal effect, it accounts for probabilistic ($p<1$) outcomes (label ***6***) which can be leveraged to compute discounted objective values over time, which in turn can be useful for policy iteration.

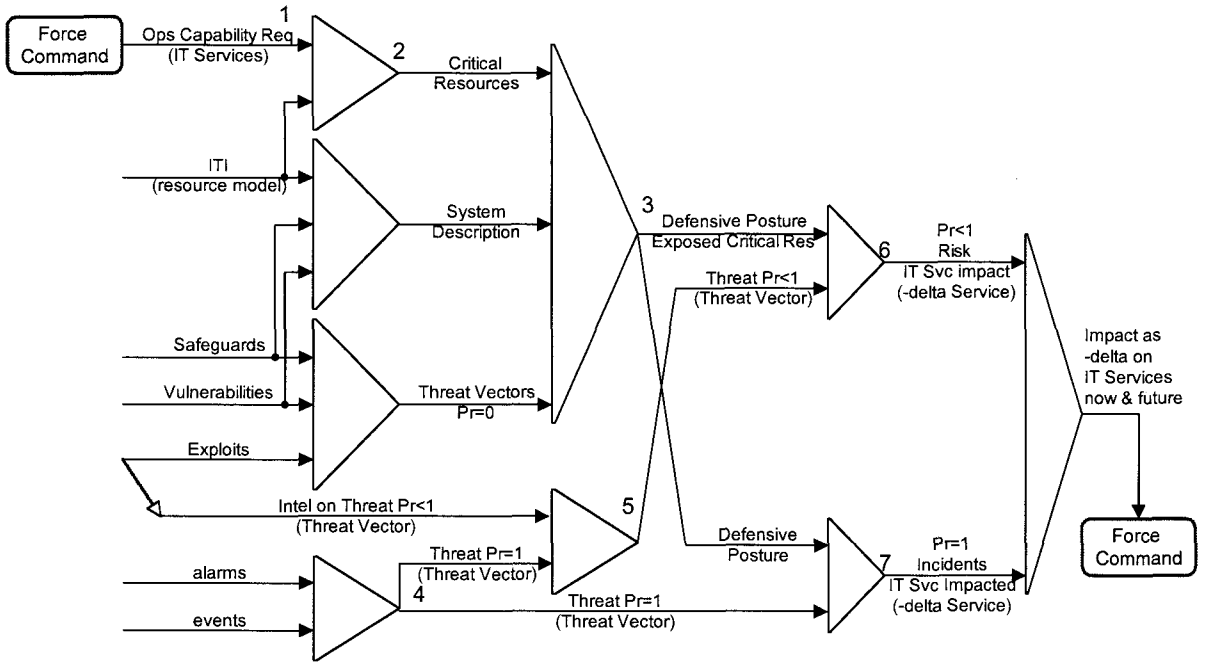

Figure 2.7 Computer Network Defence Situation Awareness model (Lefebvre et al.)

Today's operational approaches to reducing risk of IT systems are mostly compliance-driven and audit-based, as we pointed out in the introduction (Chapter 1). One such common approach is known as Threat and Risk Assessment (TRA) and refers to a manual review of network assets risk exposure. The Communications Security Establishment Canada (CSEC) TRA process shown in figure 2.8 provides a useful framework for policy management automation. This control loop presents three decision points: design risk, implementation risk (certification), and change risk (maintain accreditation).

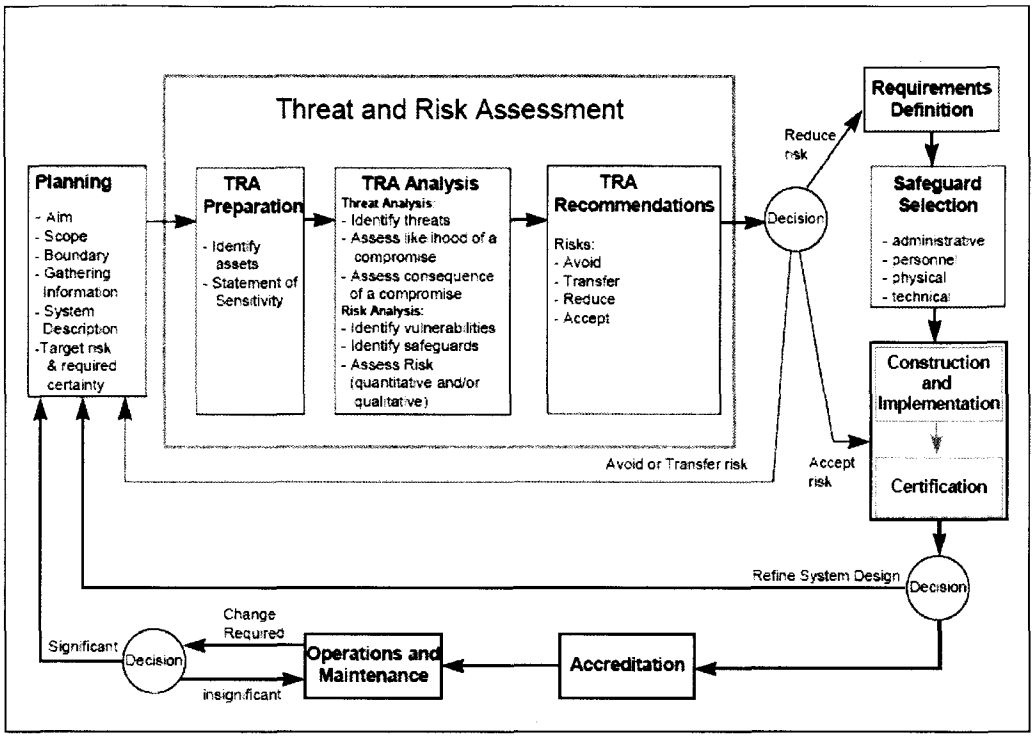

Figure 2.8 Communications Security Establishment Canada Threat and Risk Assessment Methodology[4].

This type of operational model does not account for the response capacity, or resources. Furthermore, the complex relationships between assets and safeguards should be accounted for. An automated security policy management system would require that this sort of TRA process be performed dynamically, and automatically. Intuitively, the speed at which an organisation can iterate through this cycle should also affect its overall risk exposure.

In general, automation of security policy management aims to reduce the risk to which organisations are exposed. We surveyed a number of different approaches but none have been demonstrated over real, or even realistic, environments. Mostly, this is caused by their inability to adapt to complex CND situations and to dynamically account for business risk impact. These observations support our desire to investigate Autonomic CND using an adaptive controller with different policy representations, as well as a risk metric accounting for both business needs and security events.

From our review of prior work in the fields of Autonomic Computing and Automated Security Policy Management, we derived an Autonomic CND system prototype which we introduce in detail in the next Chapter.

[4] The CSEC Threat and Risk Assessment method was first published in 1999. It was replaced in 2007 by the Harmonized Threat and Risk Assessment Methodology (HTRAM). The former is used here because it presents the process as a control loop, which is conceptually the same for HTRAM.

Chapter 3

Reinforcement Learning for CND: A System Prototype

This Chapter describes our proposed experimental framework for Autonomic CND. The resulting system prototype integrates RL with CND environment DEDS simulation and Dynamic Risk Assessment, and is designed to allow for the generation, trial and improvement of response policies.

We initially introduce in Section 3.1 the system's high-level architecture (shown in figure 3.1). We then describe in detail our DEDS simulation environment (Section 3.2) and novel Dynamic Risk Assessment algorithm (Section 3.3). We conclude this chapter with the RL algorithms implemented and the associated state representation and reward functions (Section 3.4).

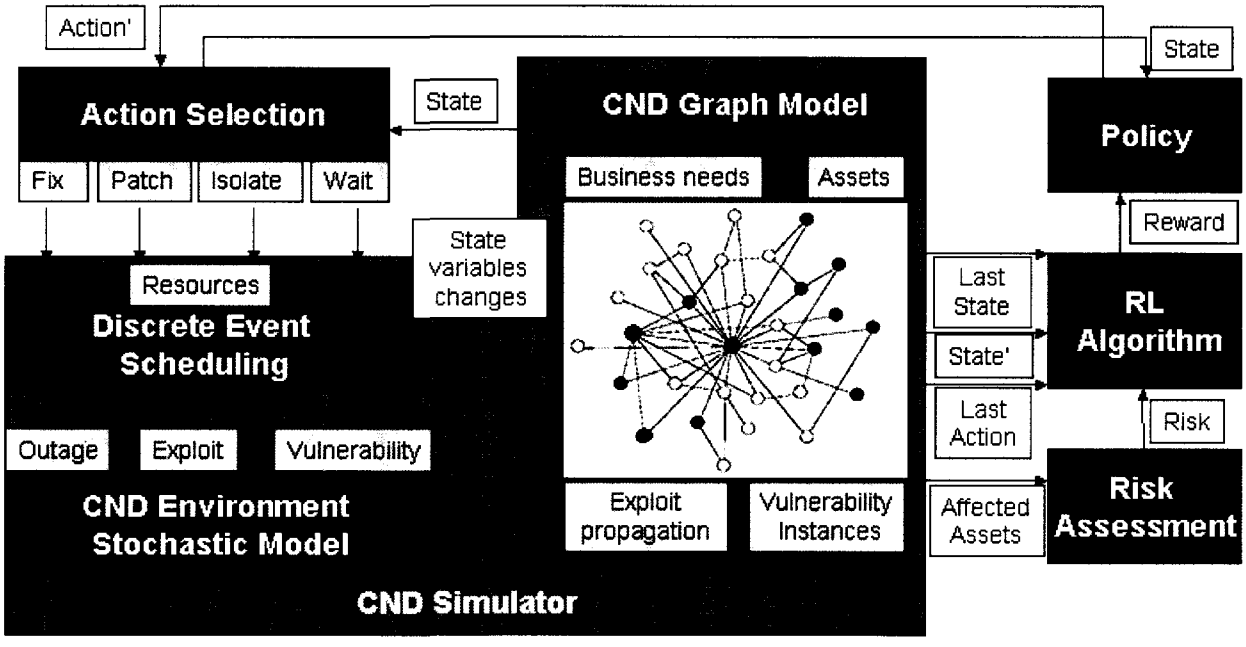

Figure 3.1 Architecture of an Autonomic CND system using RL and risk states.

3.1 Autonomic CND Experimentation Framework Architecture Overview

The Autonomic CND experimentation framework of figure 3.1 is broken down into seven main modules, shown in dark blue: *Action Selection, Discrete Event Scheduling, CND Environment Stochastic Model, CND Graph Model, RL Algorithm,* and *Risk Assessment.*

Action Selection: For every system state change, the *Action Selection* module searches the *policy* for the best next action, *action'*, to implement given the current *state*. This is performed either through a greedy search for the maximum Q-value in the *policy* or using a *softmax* Boltzman probability driven choice. The resulting *action'* (*fix, patch,* or *isolate* an *asset*, or *wait*) is then passed to a *resource* (a NOC staff) and scheduled using *Discrete Event Scheduling.*

Discrete Event Scheduling: The scheduling module generates event duration and puts the *action* event in a queue, ordered by time. In some scenarios, it also receives exogenous events from the *CND Environment Stochastic Model.* The scheduling module then advances the simulation clock to the next event in the queue and communicates the associated *State variables changes* to the *CND Graph model.*

CND Environment Stochastic Model: This module generates arrival times for *outages, vulnerabilities* and *exploits* according to predetermined probability distributions.

CND Graph model: The *CND Graph model* keeps track of the status of each *assets* and their interdependencies and link to the *business needs.* It also enforces the rules for the creation of *vulnerability instances* and *exploit propagation*, considering safeguards.

Risk Assessment: The *Risk Assessment* module queries the *CND graph Model* for the list of *affected assets*, computes the *risk* R(t), then passes this scalar to the *RL algorithm.*

RL algorithm: Before each *action* implementation is completed, the *RL algorithm* queries the *CND Graph Model* for the current *state.* After the *action* implementation, the RL module queries the Graph Model for the new *state'* and updates the *policy* with these quantities and the associated *reward* $\Delta R(t)$, or the integral of R(t), depending on the training strategy.

Policy: The *policy* updates its state-action map with *rewards* and *states* received from the RL algorithm. It also provides the *action selection* module with the preferred action for a given state, based on its Q-value mapping.

3.2 Simulation of the CND environment

Discrete event dynamic systems (DEDS) simulation uses scheduled events and a finite set of system states to describe a system's behaviours. This approach is useful for modeling complex human-made systems and has been extensively used in similar research as pointed out in Chapter 2. Simulation with DEDS leverages probabilistic models to schedule events in continuous time space. These events, in turn, change the system state at various discrete times. A clock advance algorithm is leveraged to only evaluate the system at these times, which greatly improves performance.

The objective of our DEDS simulation implementation is to measure system risk evolution on a computer network environment as various events and actions take place. The events must include common CND situations, namely *outages, vulnerabilities* and *exploits*. The actions must include typical CND responses such as *fixing, patching, isolating* and *waiting*. These actions are performed by a limited number of network operations centre (NOC) staff.

When simulation is used to support decision making, which is usually the case, the fitting of the probability distributions is essential to establish the accuracy of the measurement: "Garbage in, Garbage out", as they say. In this research effort however, the constraints on the fitting of these distributions is merely in order to represent the complexity of the underlying environment. Our intent is to demonstrate viability of some algorithms to find an optimal policy, and not to find the actual real-world optimal policy, which would require a much more rigorous validation and verification process.

We investigated the stochastic properties of CND externalities/events: exploits, vulnerabilities and outages. The concept of *arrivals* is common to these events. Considering a continuous time space, the probability of having an event at a given value of time **t**, is very low. By subdividing the time intervals into **n** discrete slots, and letting **n** go to infinity, we can model arrivals by a binomial distribution with parameter **p**, the probability of seeing an event for each time slot. Since **p** is very small and the probability of seeing two events in the same time slot is negligible, this model converges to a Poisson distribution (proof in [64]). It can be shown that inter-arrival times, or the number of time slots between two events in the discrete case, tend towards an exponential distribution. This result is important in simulation because it allows events to self-generate, that is, schedule the next event when processing the last event. Poisson distribution has been shown to accurately model various environment, including car traffic, customer arrivals, and packet routing. The demonstration that this distribution is also suitable for vulnerabilities, exploits and outages has not been made conclusively, although there are some efforts with this respect [55]. Nevertheless, we will assume the exponential distribution captures the complexity of the inter-arrival rate between CND events, and we will focus on assessing its parameter lambda (λ), or mean.

Vulnerability probability distribution

We considered two attributes of vulnerabilities to assess their mean inter-arrival rate: the vulnerability disclosure rate, and the affected product distribution. The first is affected by the global capacity of the IT community to find, document and share vulnerability information. The second is affected by the variation of this capacity amongst the different IT products, which is influenced by market share and product category.

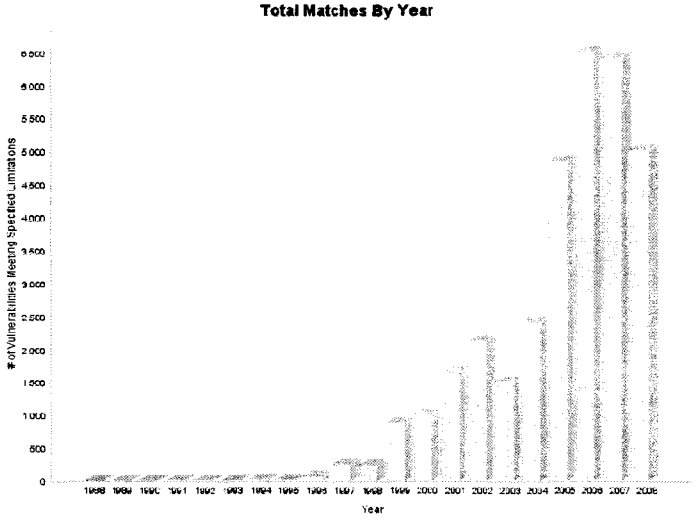

Figure 3.2 Yearly disclosure of vulnerabilities by NIST NVD.

The US National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD) is the largest and most trusted repository of vulnerabilities in the world. It counts more than 33,000 entries. In 2006 and 2007, approximately 6,500 vulnerabilities were documented (for an average of 0.74 vulnerabilities per hour). These vulnerabilities affected approximately 50,000 different products. This represents a mean of 0.13 vulnerabilities per product per year. Clearly, this is not a valid model. First, the product list includes older versions of software which are now rarely found (ex: Windows 3.1). Second, these products have different size and level of complexity, making them more or less subject to vulnerabilities, as supported by [1] and referred to in the software industry as software defect density metric[5]. Finally, the market share of these products skews their probability distribution. As an example, in 2008, Linux Enterprise from Red Hat[6] had 4 disclosed vulnerabilities versus Microsoft XP[7] which had 59. Considering that

[5] A 2008 open source software study by Department of Homeland Security showed that popular products had on average 1 defect per 4,000 source lines of code.
[6] Red Hat Enterprise has approximately 30 millions lines of code.
[7] Microsoft XP has approximately 50 millions lines of code.

Microsoft XP has nearly 90% of market shares, compare to 4% for Linux[8] certainly plays a role in skewing this vulnerability-product distribution [1]. This reasoning also applies to client-side applications such as browsers. One should note however that although Internet-Explorer 6 has 78% of the market, it had the same number of vulnerabilities as Firefox 1.5, which accounts for 8%. It should also be mentioned that close to 30% of all vulnerabilities have public exploits, and that nearly 60% of of these vulnerabilities are for client browsers (as shown in Figure 3.3 and 3.4). For routers, the code base is smaller than for operating systems because functionalities are limited and are partly implemented by the appliance (hardware). This should lead to less vulnerability by applying the software density rule. As an example, main Cisco router models have less than one vulnerability disclosed by NIST per year on average. However, the recent venue of software-based routers[9] running on common operating systems may change this soon.

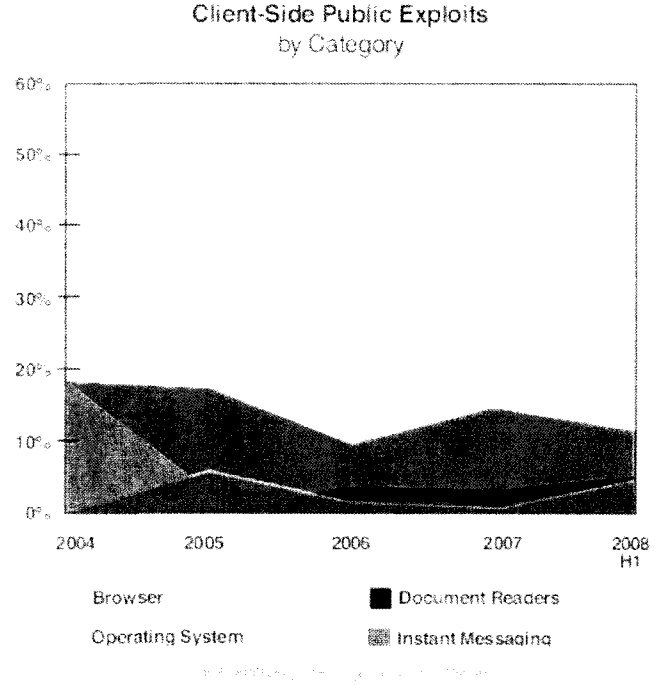

Figure 3.3 Vulnerability with public exploit per product type[10].

[8] http://marketshare.hitslink.com
[9] http://www.experiencefestival.com/a/Router_-_Software_routers/id/5429596
[10] Reproduced from IBM X-Force 2008 Mid-Year trend report.

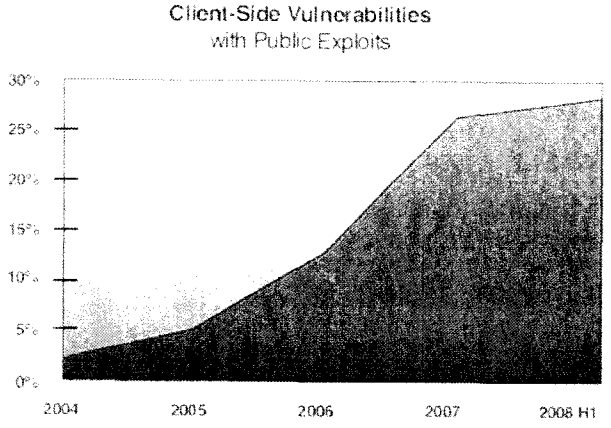

Figure 3.4 Vulnerability with public exploits[11].

As a result of these observations, we derived the estimate of the mean vulnerability inter-arrival rate by assuming a flat distribution across our synthetic environment product line using the worst case scenario for exploit availability (browsers), and by assuming our test infrastructure covers 10% of the entire affected product list, which implies some level of standardization. This yields the following result:

$$\lambda_V = 0.3 * 0.6 * 6500 * 0.1 / 365 / 24$$

$$= 0.0134 \text{ Vulnerabilities per hour}^{12}$$

Exploit probability distribution

The probability that an exploit is available for a disclosed vulnerability is also a challenging quantity to assess. Indeed, there are indications that the "known world" of exploits disclosure represents only a fraction of the total number of exploits. The reason is the growing black-market for such exploits, providing secrecy incentives[13] and making there "free" release much less likely [83]. Furthermore, a study by IBM research shows a growing number of disclosed vulnerabilities having exploit code available the same day.

[11] Reproduced from IBM X-Force 2008 Mid-Year trend report.
[12] Non public data sources confirm this approximation for reasonably large IT infrastructures.
[13] A virus for a public vulnerability could be bought in 2007 for US$1.37. Zero-Day exploit are available for a few hundred dollars.

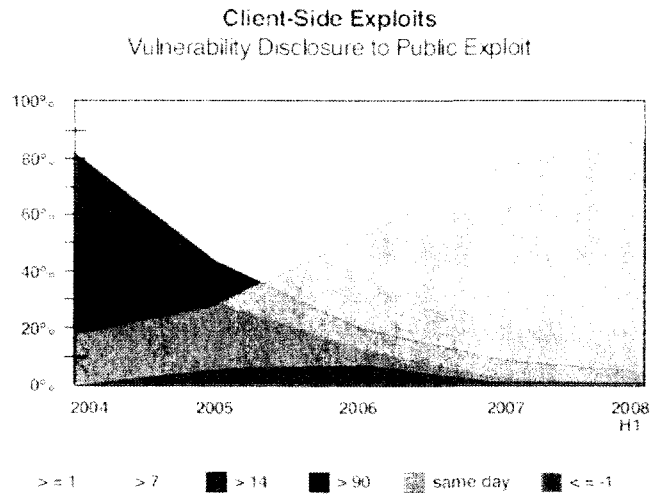

Figure 3.5 Trend showing the increasing availability of a public exploit on same day the vulnerability is disclosed[14].

As a result of these observations, we determined that approximately 80% of the vulnerabilities should be exposed to public exploit within 24 hours. We assumed that once an exploit is made public, an attack will be launched every time on the simulated environment. Therefore, we have:

$$e^{-\lambda_E * 24} = 0.8$$

$$\lambda_E = 0.0093 \text{ per hour per vulnerable host}$$

Outages probability distribution

Most telecommunication equipment is designed with sound quality metrics including MTBF of several years and availability in the order of few minutes downtime per year. These metrics however often exclude environmental factors such as maintenance activities which may affect the operation of these systems. These include upgrades, power system maintenance, configuration errors, and physical damages caused by externalities (floods, air conditioning failures, etc.). As a result, many of the telecommunication service outages are self-inflicted [34] [78] and could be avoided.

[14] Reproduced from IBM X-Force 2008 Mid-Year trend report.

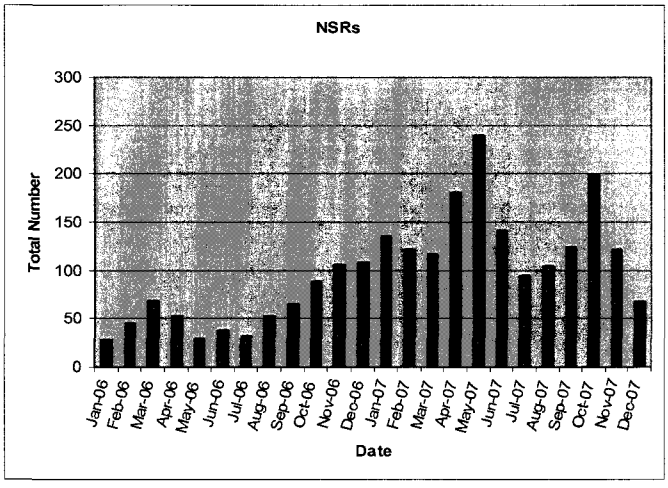

Figure 3.6 National Service Release at the CFNOC for 2006 and 2007.

Releasable information from the Canadian Forces Network Operations Centre (CFNOC) showed that in 2007, approximately 1645 forecasted outages for maintenance purposes were scheduled (0.1878 per hour on average). These are referred to as National Service Releases (NSR) and only include maintenance affecting the national infrastructure, and therefore, do not account for maintenance performed by local system administrator. The year 2007 has an unusually high number of these NSR due to a major backbone carrier change[15]. A portion of these NSR is however related to patch and upgrades of existing equipment in order to mitigate existing vulnerabilities. We estimate these to account for 20% of all NSR. Finally, other non-public data source allow us to assume that there are actually more equipment failures that manufacturers' specifications may indicate. In a 2 months period, 298 events of unreachable hosts were recorded on a given infrastructure. Most of them required manual reset of the equipment.[16] Finally, the number of these outages is directly influenced by the size of the infrastructure. The exact number of monitored devices could not be obtained but we can realistically assess that each site has approximately two backbone assets, hence approximately 1012 hosts.[17]

$$\lambda_O = \frac{\frac{1645}{1.2} + (298 * \frac{12}{2})}{365 * 24 * 1012}$$

$$\lambda_O = 0.00036 \text{ per hour per host}$$

There are various DEDS software packages available, both commercially and in the open-source community. Since our objective was to integrate DEDS with other technologies such as

[15] In 2006, there were 686 network service releases recorded at the CFNOC.
[16] Backbone routers have been known to require periodic reset. This was sometimes attributed to transmission layer instability, such as satellite links.
[17] A list of CF sites had 506 entries.

Reinforcement Learning, we needed a solution with a compatible interface. The University of Ottawa recently published a DEDS framework called ABCMod [10] which was made available for this research as a free, open source, java library, and met our integration requirements. A detailed conceptual description of the simulation model we implemented can be found in Annex A.

3.3 Dynamic Risk Assessment Algorithm

In order to perform dynamic risk assessment, we must provide a mechanism able to re-evaluate network risk at every state change. We define risk for this purpose as a weighted function of possible damages with respect to their likelihood, similar to Moitra's survivability metric [55]. This section describes how we propose to evaluate CND risk in our autonomic CND system prototype.

Initially, the CND environment is modeled as a three (3) layer graph. Each layer is described in table 3.1 and shown in figure 3.7.

Name	Description	Graph data source
Logical Graph (G_L)	Systemic dependencies and operational dependencies of nodes on the network. Many relationship are in a single direction. Ex: Email client requires email server.	Applications and systems managers (LCMM, LCAM, FASM, etc), "6"-staff (military), CIOs (civilian), specialized sensors for Web Applications dependencies (IBM TADDM, BMC discovery, etc.), traffic analysis statistics (NetFlow).
Physical Graph (G_P)	The telecommunication network which enables packet exchanges between vertices. Most of these connections are bidirectional. Ex: A fiber link between two routers. All logical assets must be in the physical graph as well. The inverse is not true.	Network engineers and administrators, topology discovery tools, routing tables.
Affected Physical Graph (G_A)	A copy of the physical graph with the initial vertex set being affected by events. Ex: Asset A has an outage, so all its edges are removed.	System generated at run time.

Table 3.1 The various Asset graphs used in the CND model.

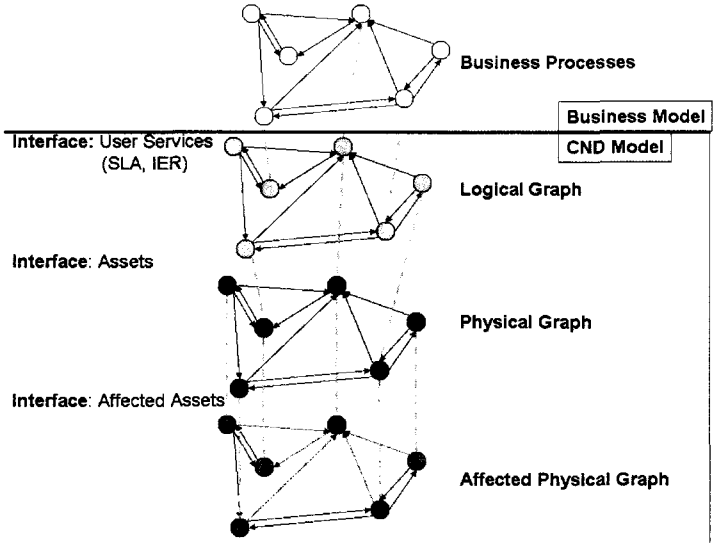

Figure 3.7 The CND model using three graphs and interfaces to the business processes graph.

These in-memory graphs are then used to compute a value for each asset. This value must be traceable back to a business need, as argued by Lefebvre[50] and Wang[77]. An Asset will inherit the value from another Asset if it is enabling it to fulfill its function. Although Markov Chains could be used to perform this inheritance, as proposed in [55][64][66], we are interested in a deterministic approach allowing for empirical assessment and which may be used in the future as a benchmark.

In the logical graph, edges represent "functional requirements", which translates into "communication requirements" through the physical graph. Special care must be taken to avoid counting the same value twice, or worse, in cyclical fashion. A simple test for this is to assure that no Asset value is greater than the sum of all User Services values. The algorithm also tracks every paths supported by each Asset, and splits the Asset value into a set of contributions towards each User Service. The Asset value set **Av** is composed of User Service values ($\mathbf{US_i}$) such that: $\mathbf{Av} = \{\mathbf{US_1}, \mathbf{US_2}..\mathbf{US_n}\}$, where **n** is the number of User Services, and $0 \leq \mathbf{US_i} \geq 1$, and the resulting asset value $\mathbf{a_v}$ is given by:

$$a_v = \sum_{i=1}^{n} US_i \,, \qquad \text{where} \quad US_i \in A_V$$

The following pseudo-code describes the update algorithm for the elements of **Av** for each Asset:

1. For each logical graph $\mathbf{G_L}$ edge, find all paths in the physical graph $\mathbf{G_P}$ connecting the source and target Asset. (greedy, depth-first search)
   a. For each paths enabling edge $y \in G_L$:
      i. Store a path unique identifier in the **PathID** set of each Asset on this path, and
      ii. Store the same identifier in the **USi USPathID** set.

2. For each asset:
   a. For each **USi** in **Av**:
      i. Compute the ratio of the number of unique paths supported $\mathbf{P_{y,s}}$, over the total number of unique paths for the logical edge y, $\mathbf{P_{y,total}}$:

$$R_{US_i} = \frac{P_{y,s}}{P_{y,total}}.$$

      ii. Update $\mathbf{US_i}$ using logical edge **y** source Asset:

- Compute: $Av^*_{US_i} = Av_{source\_US_i} + R_{US_i}$,
- IF $Av^*_{US_i} > Av_{US_i}$, then

$$Av_{US_i} \leftarrow Av^*_{US_i}$$

3. Repeat 2 until no Asset value is updated.

Table 3.2 Asset value vector update algorithm.

The final step of the dynamic risk assessment algorithm consists in computing the expected damages resulting from the affected graph G_A. We assume that the damages caused by an event are proportional to the values of the assets it affects. However, both the damages and their likelihood are modeled as functions of time. As a result, we obtain the following equation for a risk value:

$$R(t) = \sum_{i=1}^{n} \sum_{j=1}^{m} d_i(t) * p_j(t)$$

where:

- **n** is the number of affected assets
- **m** is the number of events
- $\mathbf{d_i(t)}$ is the damage function incurred by the business at time **t** for asset **i**
- $\mathbf{p_j(t)}$ is the likelihood of occurrence of an exploit for a vulnerability event **j** at time **t**; 1 otherwise (after exploit occurrence, p=1 for the targeted vulnerability).

In our algorithm, we further hypothesise that when an asset has multiple vulnerabilities $\mathbf{v_i}$, each disclosed for a time period $\mathbf{t_i}$, the resulting probability of exploit of the host increases in the following manner:

$$p_j(T) = P_{v1(t1)} + (1\text{-}P_{v1(t1)})P_{v2(t2)}) + (1\text{-}P_{v1(t1)})(1\text{-}P_{v2(t2)})P_{v3(t3)}) + (1\text{-}P_{v3(t3)}) \ldots (1\text{-}P_{v(n-1)(t(n-1))})(P_{vn(tn)})$$

where :

- $P_{vi(ti)} = e^{-\lambda_E * t_i}$, $0 \leq \mathbf{P_{vi(ti)}} \geq 1$
- **n** is the number of vulnerabilities on the asset minus the first one.
- $\mathbf{t_{vi} < T}$
- **T** = period of assessment.

Each vulnerability's probability of exploit $\mathbf{p_j}$ is affected by $\mathbf{t_i}$, the time period since its disclosure. As discussed in the exploit probability distribution section, we used an exponential model with $\lambda_E = 0.0093$.

As for the damage function $d_i(t)$, we used a linear approximation to capture the time sensitivity of the services:

$$d_i(t) = Av_i * f(t) = Av_i * (bt - a)$$

for $a < t < \frac{(1+a)}{b}$,

$d_i(t) = 0$ if $t<a$

$d_i(t) = Av_i$ if $t > (1+a)/b$ or $p_{vi}<1$

t: period without service.

The variable ***a*** represents the maximum consecutive unavailability time tolerated before damages occur. The variable ***b*** represents the time sensitivity of the business on the affected service. This is an empirical function which allows *fuzzifying* the service level required by the business.

The sum of damages $\sum d_i(t)$ presents some challenges. In the case of a single asset being affected by an outage ($\mathbf{p}=1$), $r(t) = d(t)$, where *d(t)* is the asset value ***Av*** multiplied by the time sensitivity function. But in the case of a single asset affected by a vulnerability ($\mathbf{p(t)}<1$), *d(t)* is empirically set to ***Av/2*** , as **p** increases with time. This is to limit the contribution of potential damages and avoid a discontinuity in R(t) as these damages actually occur.

When more than one asset is affected, since our model is not additive, we need to select, amongst the affected assets, which ones to sum. Since assets can contribute to different user services in different proportion, only independent contributions should be added to avoid double-counting.

We test for independence using the intersection set of paths between affected assets and *User Services*. If the intersection set is not empty (in the case of partial dependency), we only sum the asset of the set with the highest value (worst case). We perform this summation of independent asset contributions for each *User Service*, upper bounded by the *User Service* value itself.

This algorithm was implemented using a Java library called JGraphT. This library provides a scalable object model for vertices and edges as well as a number of graph walking algorithms.

3.4 Reinforcement Learning for Autonomic CND

We provided a review of Reinforcement Learning in **Chapter 2**. There are many variants of RL algorithms and a framework allowing for experimentation with several of them is desirable for our system prototype. Specifically, we are interested in the off-policy Q-Learning algorithm, because we assume no prior knowledge of what a good CND policy might be. We also want to leverage eligibility traces to improve convergence, and use function approximators to provide scalability. Many packages exist already for RL research purposes. We chose two specific packages to be integrated into our Autonomic CND system.

The first package selected uses table policies without function approximators. It was originally developed by Time Eden, Anthony Knittel and Raphael van Uffelen at University of New South Wales, Australia. It includes Q-Learning (off-policy), SARSA and Q(λ) (on-policy) algorithms, and both soft and greedy selection methods. The java library was initially developed as part of a demonstrator for a cat-and-mouse grid world, as shown in figure 3.8. We significantly modified this package to interface with the DEDS simulation and the CND graph model, but retained the algorithm implementation.

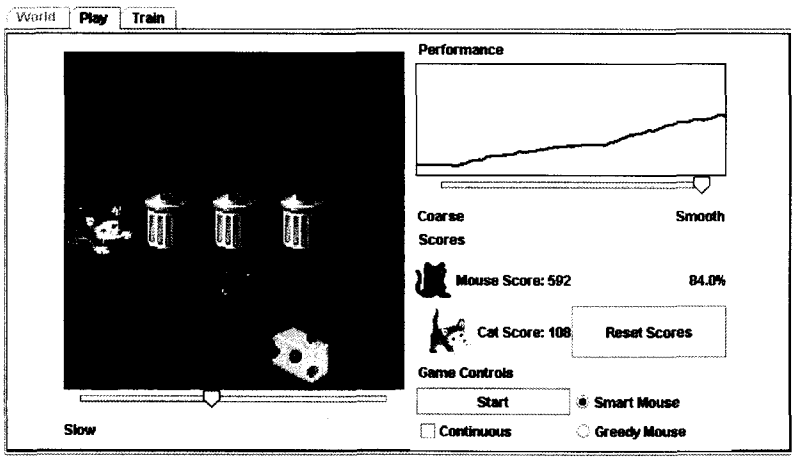

Figure 3.8 The Cat and Mouse Grid world scenario for RL table policy[18].

[18] http://www.cse.unsw.edu.au/~cs9417ml/RL1/applet.html

The second package selected is an implementation of the Q-Connectionist algorithm described in [48]. It uses multi-layer neural networks to generalise the states-actions space, update and store the policy. This java library was implemented in a lunar space-craft landing simulator. Its ability to converge in a continuous state space and for an increasing number of control actions gave us great hope of its suitability for the CND environment. The package also implements random mutation of the neural networks (similar to Genetic Algorithm) and eligibility traces.

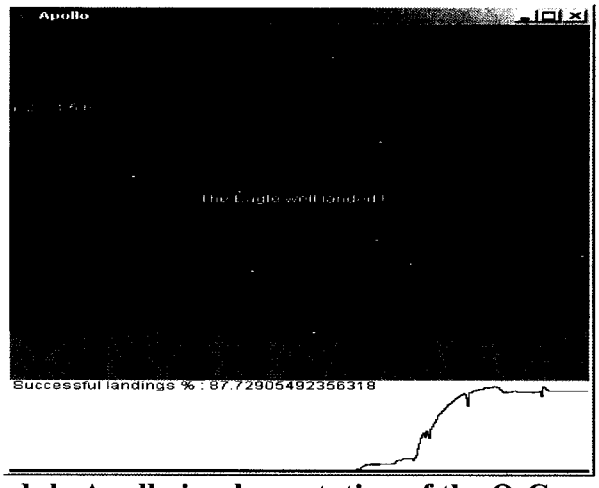

Figure 3.9 Lunar module Apollo implementation of the Q-Connectionist algorithm[19].

3.4.1 CND States and Actions space

In the CND graph model, each node-Asset has a number of states. In the simplest case, a graph state space can be modeled using two (2) states per node (up or down), yielding $\mathbf{2^n}$ states. The combinatorial growth is problematic for cases with many states per node. Each node in our CND model can have up to four (4) states (OK, outage, vulnerable, exploited). In an 11-vertices graph (we will introduce an 11 asset-scenario in **Chapter 5**), this gives 4^{11} (4 194 304) states. Brute force approaches could not scale up to real size networks (several thousands assets) at this rate. This justifies our consideration of state generalization in our architecture.

Another approach to reducing the state space is to assume that certain graph configurations are highly improbable. One such assumption could be to reduce the number of possible concurrent defective assets. Again, for an 11 vertices graph, limiting to four (4) the number of concurrent affected assets reduces the state space to:

[19] http://www.elsy.gdan.pl/

$$\sum_{n=0}^{n=4}\left(\frac{11!}{n!(11-n)!}\right)*3^n = 26730+4455+495+33+1=31714 \quad \text{states}$$

That is, four (4), three (3), two (2), one (1) or zero (0) assets can be affected at a given time. Note that the number of states per node is reduced to three (3) since the "OK" state is already accounted for. This assumes that we know perfectly the state of each asset, which is not often the case in real life scenarios, due to sensor accuracy issues discussed earlier. Nevertheless, this approach has the merit of being directly influenced by the speed at which the system can react to events. The faster the controller is, the smaller the probability of concurrent events become, and as a result, the smallest the state space is (bounded by **3n+1** in the example above).

The set of possible actions also suffers from the size of the problem space. A single action type, such as patching an asset, leads to ***n*** actions to choose from, that is, one per asset. The learning agent must pick both the right action and the right asset. In our proposed model, we have 4 possible types of action: fix, isolate, patch, or wait. We do not need a "wait" action for every host because this action has not state consequence on any assets, only on resources. Therefore, our action state space is **3*n* +1** (for one NOC staff). In the 11-asset scenario, there are thirty-four (34) possible actions. This number of classes threatens convergence significantly. We propose two approaches to loosen this constraint. The first one is to train a state generalization agent for each action rather than have a single agent learn to differentiate all action-class. The agent with the highest Q-value determines the optimal action to select (greedy search). This approach is implemented using the Q-Connectionist algorithm. The second approach consists in constraining the number of possible actions allowed in a certain states using *a priori* knowledge. Since we know that only affected assets and border protection assets should be subjected to any actions, only a subset of actions will be valid for any given state. Every other action would then never be selected and training would never need to explore these states.

3.4.2 CND Reward Function

There exist different strategies in rewarding a reinforcement-learning agent. These strategies can affect the speed of convergence and the accuracy of the optimum found. A quantity of interest is the integral of the function ***R(t)*** over the full learning episode, similar to the reward function used by Barto in the elevator dispatch problem [3]. The risk integral indicates whether a given trajectory (or decision path) is better than another. The assumption is that a business which assumes lower risk over time will be better off.

The Q-function uses a discount factor gamma between sequential state-action pair of a policy. As a result, the algorithm will naturally find the least number of steps required to get to the absorbing state, if the reward is only provided at the end. This is not necessarily equivalent to an

optimal policy in our case since all actions do not bear the same cost (resource cost is equivalent to time in our simulation, since all NOC staff are deemed equivalent). Indeed, there might be many rapid actions leading to the same result as a single longer one (that is, more steps may be better than less, depending on how long they take). The agent may even decide to wait to see if more important events arise in the short term prior to committing to the existing events. Our reward strategy must compensate for this algorithmic bias.

The end state should be when there is no risk to the network. In reality though, there is most likely always some risk. Even in our simulated environment, depending on the steady-state equilibrium between events arrival rates and the fixing/patching rates, there might be some risk a majority of the time. This may lead to a problem since at least one absorbing state is needed for convergence. We consider two different approaches to address this issue: a bonded horizon study, and a steady-state study. The bonded horizon study, in simulation, is one where the right-hand boundary of observation interval is specified explicitly or implicitly. In our case, the bounded approach means that initial events are scheduled up-front and that the simulation ends when $\boldsymbol{R(t) = 0}$. In other words, the system must learn to fix the pre-existing events in the optimal way to minimize the risk integral. The Q-function can, in this case, use a reward provided only at the end of the simulation, which can be proportional to the inverse of the risk integral:

$$reward \propto \frac{1}{\int_{t=0}^{t=end\_condition} R(t)dt}, \quad R(t) \neq 0,$$

One problem with this approach is that RL requires reinforcements even for non-optimal action sequence in order to converge. But since we do not know the optimal risk integral upfront, it is difficult to know whether we are rewarding a local or a global optimum. Different approaches were considered to limit the impact of sub-optimal policies, namely resetting the policy (randomizing Q-values) every time a new optimum is found, essentially starting the learning over. We also implemented a genetic algorithm in the Q-Connectionist case, to randomly mutate a portion of the neural networks. This approach has the advantage of keeping some of the learned behaviour. We also used random exploration of the landscape through random selection of actions a percentage of the time, bypassing the trained policy (also known as ε-*greedy*). Finally, we implemented a policy-backup procedure, allowing for backtracking if the rate of improvement, measured as a rolling-average of the epochs where optimal risk integral is achieved, within some accuracy threshold, is not sufficient. This is analogous to Policy Gradient approaches.

The accuracy and repeatability of the utility function results are important for these learning strategies to be efficient since we are comparing and keeping maximum results. This is an issue when mixing continuous and discrete algorithms. Our proposed utility function (risk integral) is

defined as continuous. However, our simulation environment uses discrete simulation, which in turn uses a clock-advance algorithm, which skips periods with no events for performance and processing efficiency. In order to evaluate the utility function in this environment, we used a discrete event called *heartbeat* to "sample" this risk function at regular intervals for each simulation run, similar to what Tesauro used in its Autonomic Computing system. This approach comes at a memory and processing cost proportional to the sample rate. Computing the continuous utility function using a separate processor thread could reduce the impact on performance while leveraging multi-core platforms. We did not implement it in separate threads for this research.

Another reward strategy, this one not requiring sampling, involves providing small rewards inversely proportional to the instantaneous risk value ***R(t)*** improvement or deterioration (as opposed to the integral, which is monotonically increasing). This strategy helps converge more rapidly by namely punishing any *wait* action. It would be reasonable to assume that Q-Learning will optimise the average cumulative reward of epochs, which should be equivalent to the integral of ***R(t)***. This is further argued by Barto in [3] for their elevator dispatch application. We must however constrain these instantaneous rewards between zero and one in the Q-Connectionist case. Therefore, the following reward function was considered:

$$reward \propto 1 - e^{\frac{-1}{R(t)}}$$

Another challenge in our CND implementation of Reinforcement Learning is that evaluating the next state is not carried out until the action execution scheduled time, possibly several decisions, or steps, away. As an example, if there are two assets affected by outages, and two NOC staff are available, the first action will be taken for State-{2 outages}, and will be assigned to the first NOC staff. The second action will be assigned immediately after to the second NOC staff; however, the state for which both these actions were selected is the same since the first outage will not be fixed for a number of hours. Some of the system state variables did change, of course, from State-{2 outages} to State-{2 outages, one of which is assigned}. In order to have a valid estimate of the state resulting from a given action, either we change the state representation, from simply the number of affected assets, to the number of unassigned affected assets; or, we split the action selection and the learner updating steps of the Q-learning algorithm, and perform the latter only after the execution of the action. In order to implement a reward strategy for immediate actions, such as the instantaneous risk improvement, the latter option provides more flexibility. However, it does not capture the "true" sequence of state-action, as shown in figure 3.10 below.

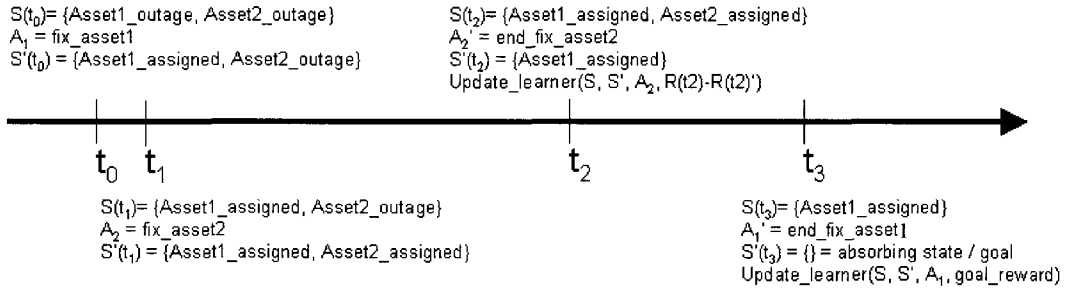

Figure 3.10 Event scheduling with action selection and learner update.

In this scenario, we can see that action $\mathbf{A_1}$ is selected for state $\mathbf{S(t_0)}$ = {outage, outage}, but the updated state-action-reward actually are {outage} and {} (goal state). We interpret this variation by the complexity of the simulation environment which can change states significantly between the time the action was chosen and the action being executed. As a result, only the snapshot of the system before and after action execution should be meaningful to the learner.

A significant challenge with the proposed Autonomic CND system prototype is its evaluation. As the system's modules interact in dynamic scenarios, data points must be collected to determine whether the system is stable given the computed control risk metric and whether the policies learned represent real improvement over other arbitrary policies in the same scenarios. The next Chapter describes the experimental set-up and the dynamic scenarios used to evaluate our proposed system prototype and its algorithms.

Chapter 4

Experiments

In this chapter, we describe the experiments used to evaluate the autonomic CND system prototype proposed in **Chapter 3**. We first describe in Section 4.1 a simple implementation of the CND graph model previously introduced. This graph captures the test environment used in follow-on experiments with its underlying IT infrastructure, set of business requirements and logical relationships between assets. We then introduce in Section 4.2 five dynamic events scenarios. These scenarios include various mixes of outages, vulnerabilities and exploits and are used to create risk conditions and support reinforcement learning agents' policy iteration. We conclude **Chapter 4** with Section 4.3 which describes the methodology used to collect and analyse data samples.

4.1 CND Test Environment: Simple 11-Asssets Case

In **Chapter 2**, we described various test environments used in similar research. The environments were generally of small size (less than 50 nodes). This condition appears necessary for results interpretation, due to the complexity and novelty of these fields. As we will demonstrate in our experimental results, even relatively simple environments can generate very complex behaviours. Some desirable characteristics of a CND test environment for our purposes should be:

- Stated business needs, captured as *User Services* assets with their relative business importance (value);
- Logical, systemic dependencies between assets, to account for n^{th} order effects;
- Several overlapping paths supporting assets dependencies, to account for redundancies and partial disruptions;

We designed our test environment to account for each of these characteristics. These allow for computing individual asset values (business importance), using the ratio of service paths supported by an asset (vertex) for each business need, as discussed in **Chapter 2**. In turn, asset values are necessary to evaluate actual and potential damages, used in the dynamic risk assessment algorithm introduced in **Chapter 3** and used to reward or punish the policy learning agent.

Our proposed environment is intended to depict a small business relying on email services between two sites, as well as on internet browsing at one site, for performing its daily operations. The proposed environment counts eleven (11) assets, or vertices. There are four interconnected sites, each with a router, to create multiple service paths. One site hosts a DNS server, one provides access to the internet gateway, two other sites each host an email server and an email client.

Three (3) assets are defined as *User Services*. They form an interface between the network topology (physical graph Gp), and the stated business needs, as introduced in **Chapter 3**. The *User Services* have the following arbitrary values, representing their importance to the business: 0.4 for *email-2*, 0.3 for *email-5*, and 0.3 for *browser-3*. One way to interpret these values is picturing them as percentage of the business Value Delivery Chain, as described in [6].

In addition, assets have directional interdependencies. Some are physical, shown as bidirectional connections in graph G_P (figure 4.1), and some are logical, representing functional relationships and shown in graph G_L (figure 4.2). The later graph includes dependencies of both email clients on their local server and one another, as well as dependencies of one site's browser client on the DNS and the WWW gateway (*Router-4*). As explained in **Chapter 3**, for the system to be fully functional and fulfill business needs, all logical relationships defined in G_L must be satisfied by connectivity through G_p.

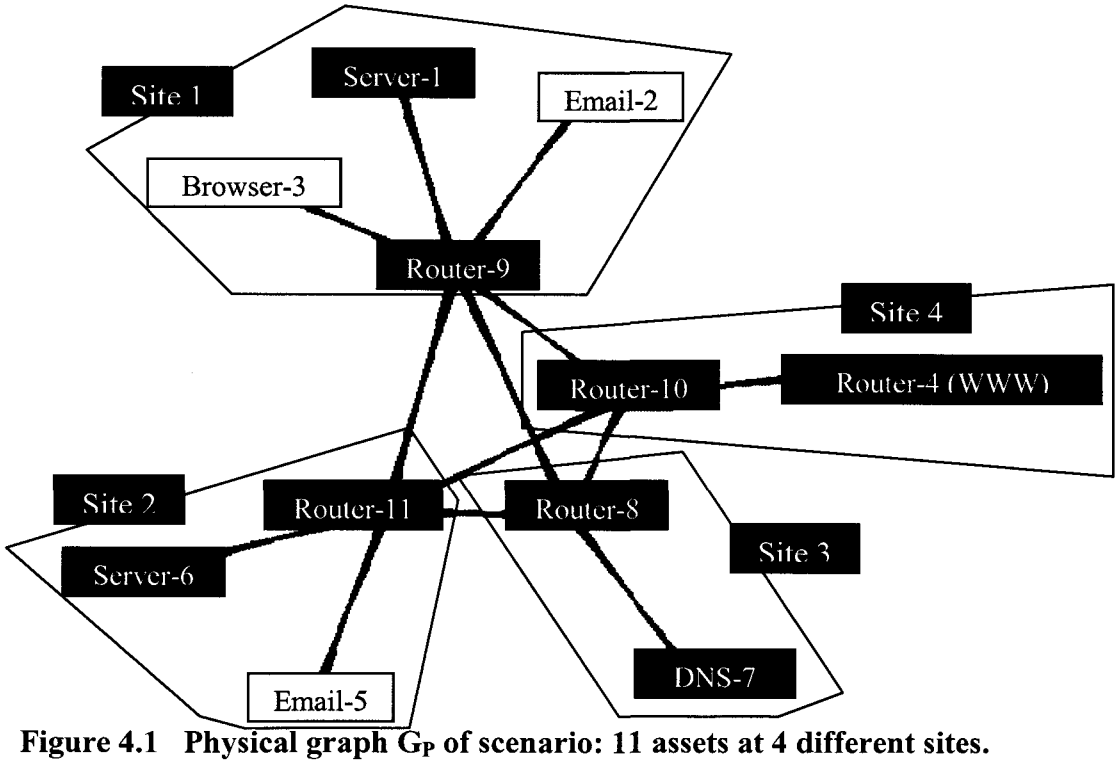

Figure 4.1 Physical graph G_P of scenario: 11 assets at 4 different sites.

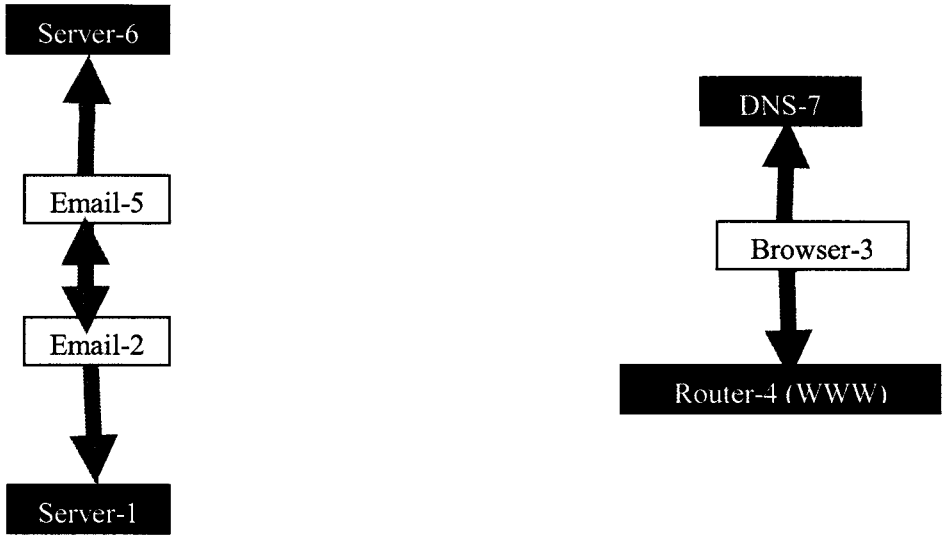

Figure 4.2 Logical graph G_L, showing two disconnected functional sub-graphs.

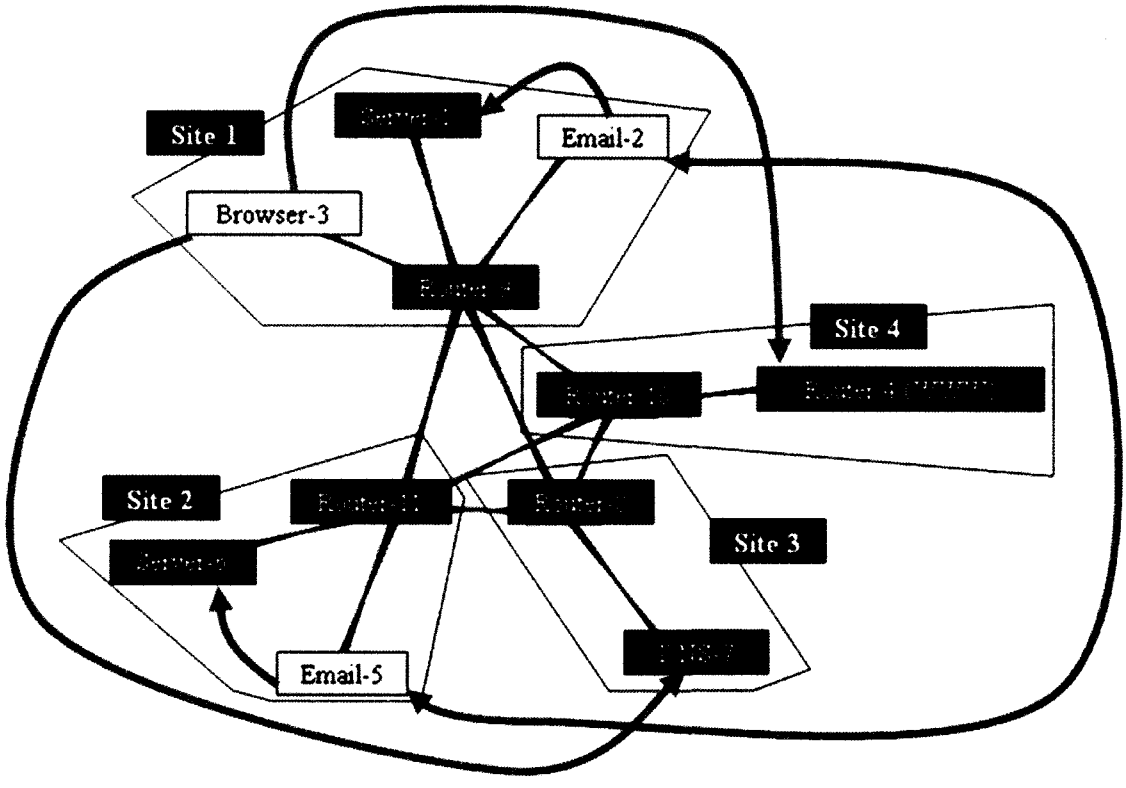

Figure 4.3 Mixed graph including both physical (blue) and logical relationships (orange), with user services (light green)

4.2 Scenarios

There are an infinite number of CND situations we could investigate using the simulation environment described above. We will limit our investigation to two specific CND problems: finding the best sequence of actions to recover from given risk situations, and continually improving a policy, on-line, as experience is gained. These problems represent different learning tasks. The action sequence problem is very similar to the maze problem introduced in **Chapter 2**, for which it is conceivable that an optimal solution be found. The continuous improvement problem, on the other hand, creates a very large number of state transitions and a generally good policy might be all that is achievable, as discussed in **Chapter 2**.

We propose to investigate five (5) scenarios which relate to these two CND problems:

Scenario 1: Simple repair sequence of four (4) outages: This simple scenario tests the general ability of the system to converge to a desired state. There are four (4) of the eleven (11) assets under outages at t=0, as shown in Figure 4.4. The learning task consists in selecting the optimal order of repair out of the 4! (24) decision branches. The learning agent also has to learn that the "wait" action is not useful in this scenario. Only one Network Operation Centre (NOC) resource is employed in this scenario.

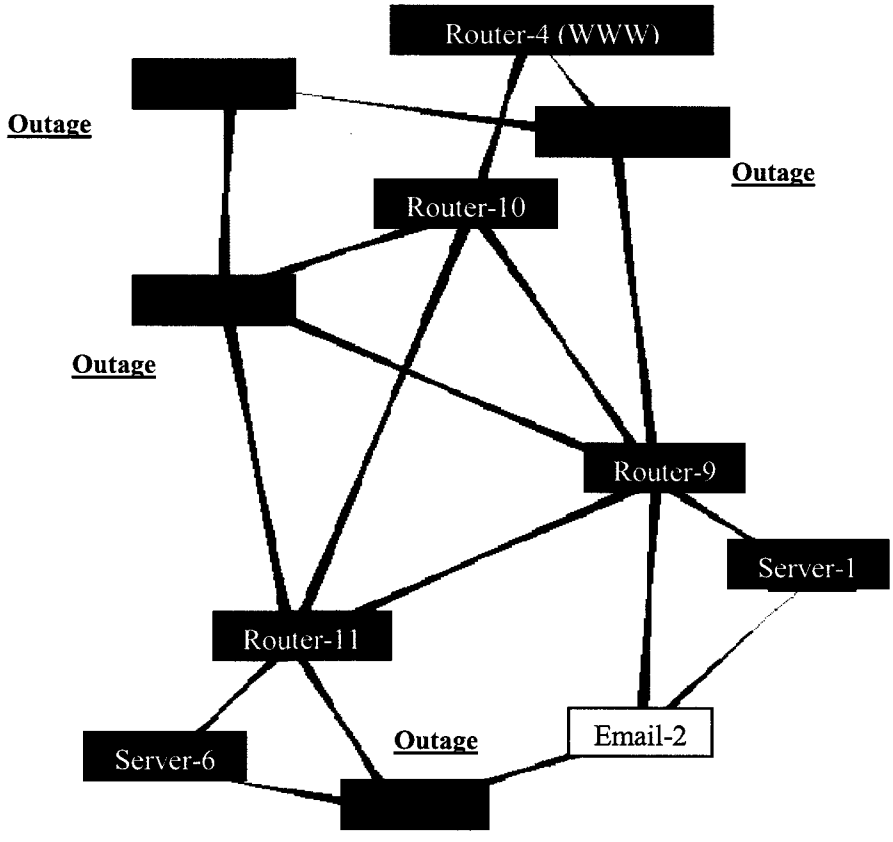

Figure 4.4 Scenario with four initial outages

Scenario 2: Simple repair sequence of eleven (11) outages: This scenario repeats the previous one but with all eleven (11) assets being affected by outages at t=0 (see Figure 4.5). For this scenario, the "wait" action is not used. The learning task consists in learning an optimal fix sequence out of the 11! possibilities (39.9 millions). The environment has a state space size of 2^{11}, or 2048 states, which represents all the combinations of eleven assets being either OK or in an outage state. Again, only one NOC resource was used.

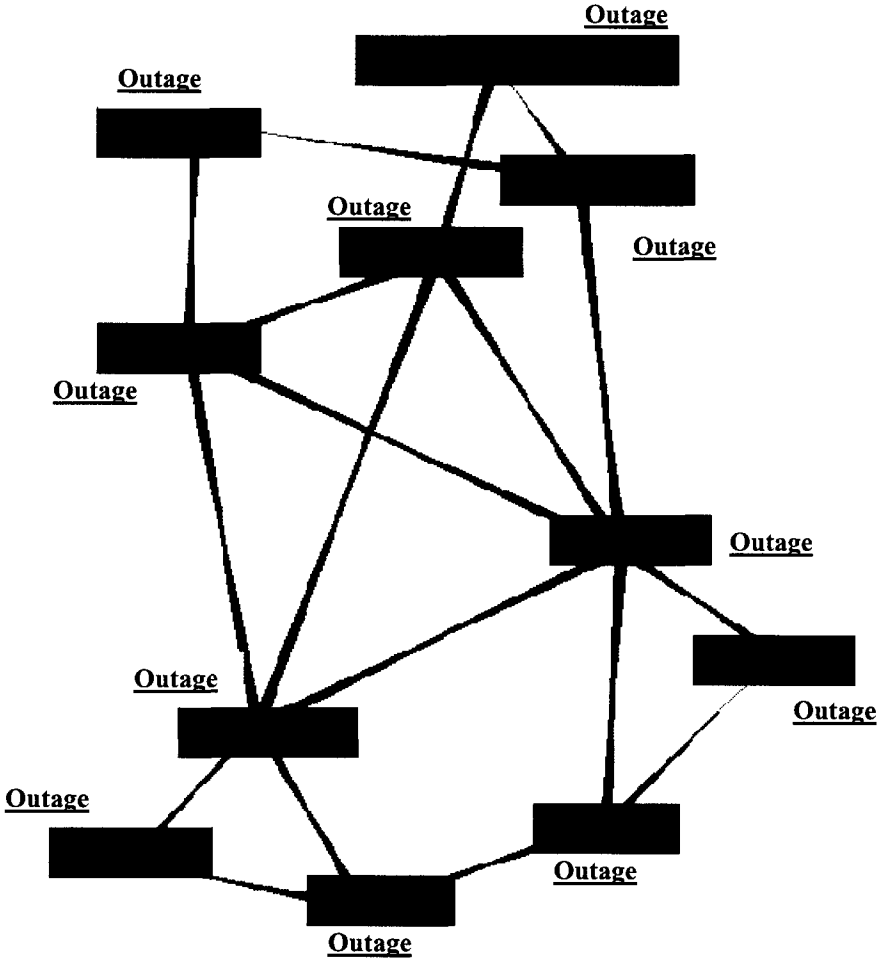

Figure 4.5 Scenario with eleven initial outages

Scenario 3: Simple patching sequence of eleven (11) vulnerability instances: This scenario consists in starting with all eleven (11) assets vulnerable at t=0. The learning task is similar to the previous outage scenarios in that it aims at finding the optimal patching sequence. One of the differences is that the risk function used for the RL reward is only affected by the computed probability of occurrence of an exploit, whereas outages only affect the actual damage (**p=1**). Another important difference is that the policy model includes *outage* states and the *fix* action for

each asset, although they are invalid states and actions for this scenario. This increases the policy size by a factor of four (4) and complicates the learner's task. No exploits are triggered in this scenario. Only one NOC resource is used.

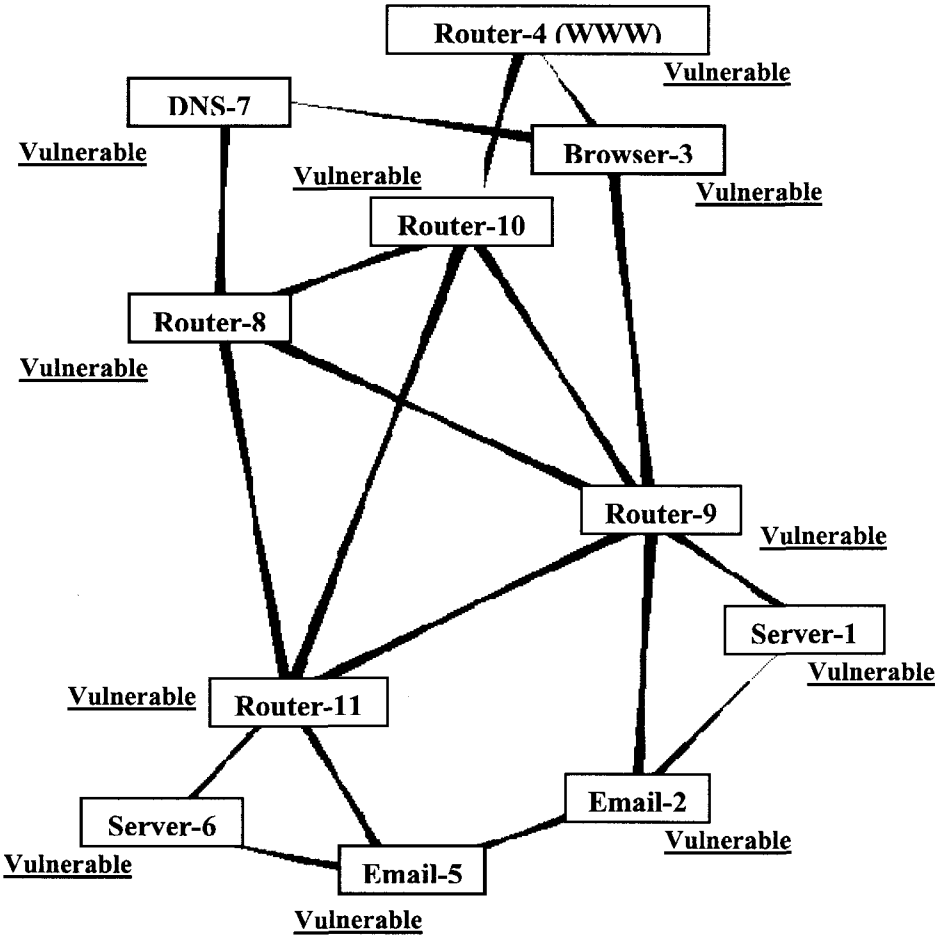

Figure 4.6 Scenario with eleven initial vulnerabilties, without exploits.

Scenario 4: Patch assets affected by a new vulnerability or fix exploits: This scenario starts with a new vulnerability at t=0. This vulnerability affects a randomly selected product (or group of asset) of the test environment (routers, DNS, emails, servers, or the browser). This new event, in turn, generates a number of vulnerability instances, depending on the number of assets of this product type. An example for this scenario is presented in Figure 4.6, with an initial vulnerability affecting the *server* product. Exploits are scheduled for each vulnerability instances, initially coming from the WWW gateway (*Router-4*), but then propagating from the inside to all other vulnerable assets at a rate empirically chosen at 4λ, that is, four times the rate of exploit arrivals from WWW gateway. Once an asset receives an exploit, its status becomes the same as an

outage. The learning task consists in patching the assets optimally, or once exploited, restoring them (fix). Only one NOC resource is used.

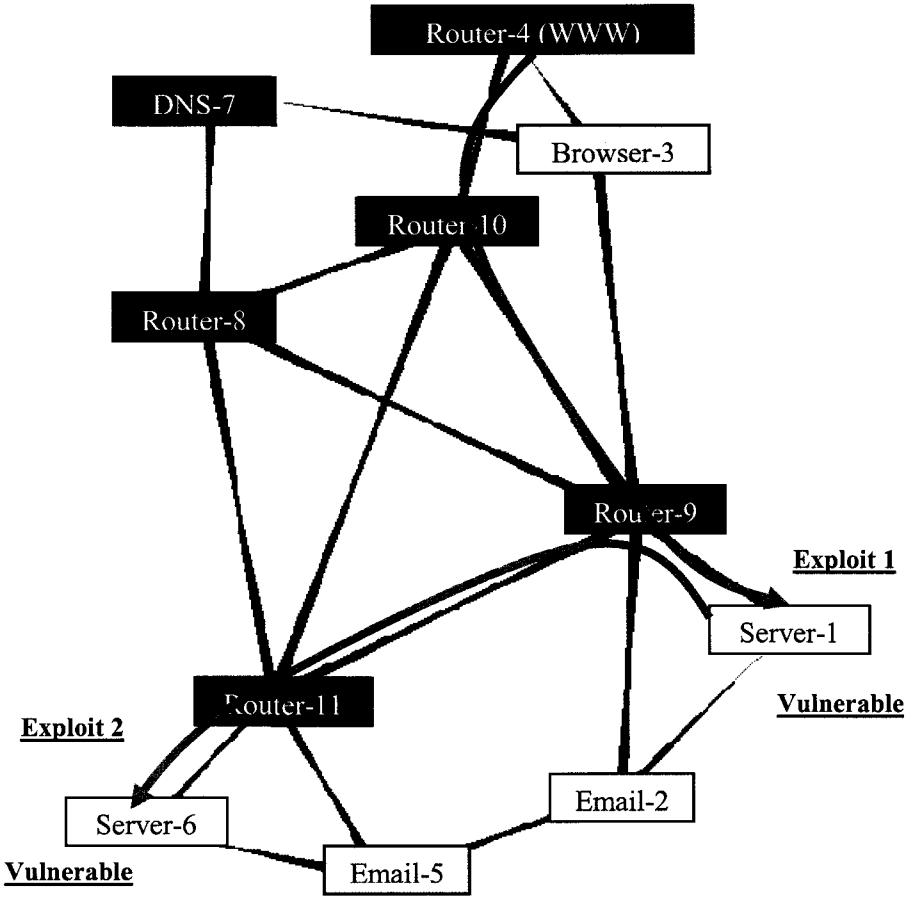

Figure 4.7 Scenario with random product vulnerabilities followed by spreading exploits.

Scenario 5: Patch, fix, isolate or wait for continuous arrival of vulnerabilities, exploits and outages: This scenario consists in letting the outage, exploit and vulnerability events arrive according to the predetermined distributions described in **Chapter 3**, in a continuous manner. All actions and all states are enabled. An example of this scenario, with an isolated vulnerability on *browser-3*, and outage on *Router-8* and a vulnerability being exploited on both of the *Server-1* and *Server-6*, is presented in Figure 4.8. The agent's learning is performed on-line, as events arrive. Every action taken and its resulting reward contribute to the next decision and to the environment's new state. We do not use epochs for this scenario: training is performed over a long simulation run (equivalent to 10 years). We also do not use the integral of R(t) as a reward

since it cannot be evaluated until the end of the simulation run. We used the following reward update for every action:

$$\text{Reward} = \Delta R(t) = R(t-1) - R(t)$$

This reward function punishes (is negative for) a *wait* decision since R(t) is monotonically increasing between actions and no assets are removed from the affected list during a wait period.

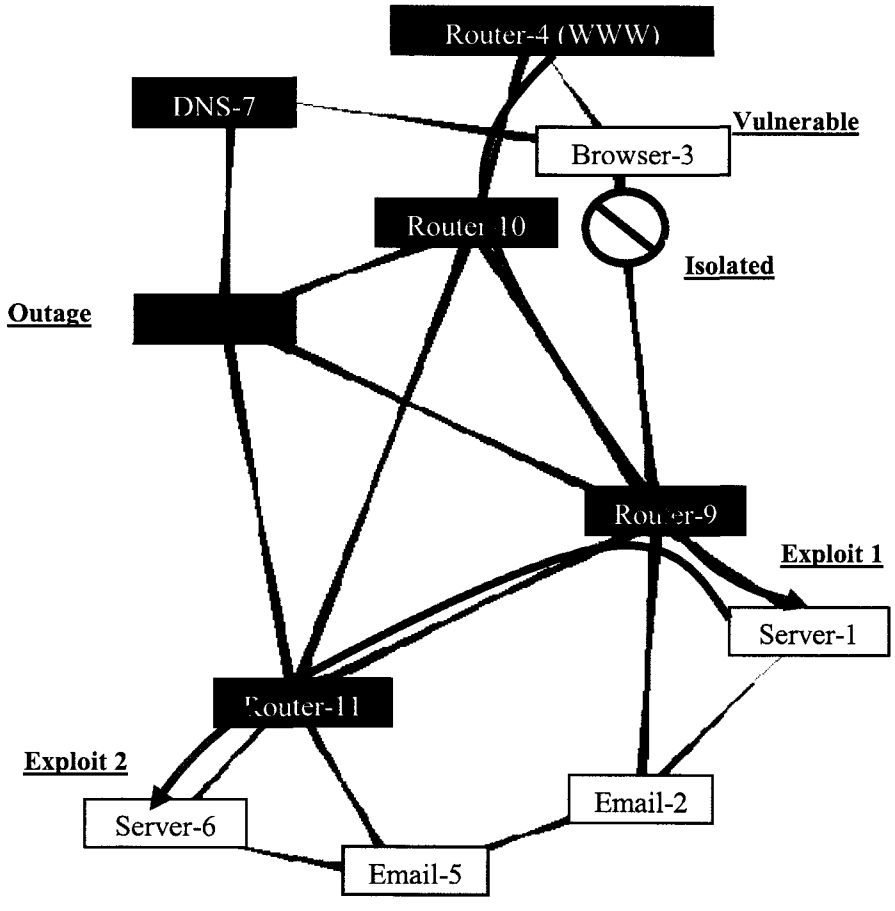

Figure 4.8 Scenario with continuous arrival of outages, exploits and product vulnerabilities.

4.3 Methodology

Our data collection and results analysis methodology has to be suitable for our discrete simulation environment and scenarios. A common technique used for DEDS simulations consists in performing a number of simulation runs for each scenario of interest, and collect data samples while preserving the random seeds used for the stochastic events generation of each run. This assures pseudo-independence between runs while allowing the exact same conditions to be

replayed for different parameters, such as different CND policies in our case. Using the Student T-distribution, we can compare sample means and confidence intervals for each trialed policy. The T-distribution is characterised by the following equation:

Student T-distribution: $\frac{Z+u}{\sqrt{\frac{V}{v}}}$

Where:

- Z is normal with mean 0 and variance 1;
- u is the sample mean;
- V is from chi-square distribution;
- v is the number of degree of freedom. v = number of runs – 1.

This distribution is convenient because it converges to the normal distribution as the number of sample increases. This is in accordance with the central limit theorem and can be used to set the number of simulation runs required to achieve meaningful results, in an iterative manner.

The confidence interval is obtained by setting the quantity ζ^*, called the confidence interval half length, and assure that the maximum displacement of the mean of the random variable, the integral of the risk $\int R(t)$ in our case, stays within ζ^*. For our analysis, we chose $\zeta^* = 0.05$. We implemented this analysis in MSExcel using the TINV function to sample the Student distribution.

For each scenario investigated, we trialed five (5) different action selection policies:

1. Do nothing;
2. Select affected asset randomly;
3. Select the affected asset with the highest value first
4. Follow the agent learner's policy:
    a. Follow the Q-Connectionist neural network policy;
    b. Follow the Reinforcement Learning table policy;

These policies were compared over the five different scenarios to assess their relative risk performance. It is important to note that the optimum risk for a given scenario is not known. We therefore rely on empirical policies such as *doing nothing* or *selecting random valid actions*, as a basis for comparison.

Another important element of our experimental methodology involves the joint roles of random seeds, simulation runs and training epochs (or policy iterations). For both the Q-Connectionist neural network (NN) policy and the Q-learning table policy, a number of training epochs must be

performed. Different strategies are used in our experiments, including training on a single initial seed, which basically preserves the same event timings for every epoch; repeating epochs on different seeds sequentially and passing the trained policy from run to run, and; repeating epochs after all runs, re-using the timing seeds of each run. The pseudo-code for the two latter strategies can be found in Table 4.1 and 4.2.

```
For i<number of runs
        For each epoch
                Use seed set i
                Run simulation
        End epoch
        Save policy for next run
End run
```

Table 4.1 Multi-run training of a policy, with epoch repeated at each run.

```
For i<number of epoch
        For each selected runs j
                Use seed set j
                Run simulation
                Save policy for next run
        End runs
        Save policy for next epoch
End epoch
```

Table 4.2 Multi-run training of a policy, with epoch repeated after all runs.

The main difference between these strategies is whether the learning agents learns iteratively a local optimum sequence of actions, for which every decision path has exactly the same action timing (that is, events such as action completion and outages always occur at the same pre-determined times for every epoch); or whether the learning agent learns a global optimum by iterating its policy over all attempted timing sequences, in a concurrent manner. In the first case, the agent is more likely to successfully learn an optimal action sequence for at least one timing sequence, whereas in the second case, the agent may not be capable of converging to a global optimum, but may be able to learn generally good action decisions for all different timing sequences.

The risk datasets are collected after the training period for each policy, using the same respective seeds, but setting the policy random action selection ($\mathcal{E}$-Greedy, used for exploration) and the learning rate, alpha, to zero (0), to assure the policy would not be modified during the evaluation runs. The simulation end time is explicitly stated (known at

the beginning) and chosen for each scenario to allow the controller to restore the risk to zero, or reach stability, depending on the scenario. The details of this methodology are further detailed in [10] and [16]. The experimental procedure is summarized in Table 4.3.

Experimentation procedure
1. Select the scenario and the policies to be evaluated.
2. Perform some simulation runs with a random policy to determine a reasonable explicit simulation end time. The end time should allow the agent to recover from the risk situation in all situations, while not being too long to limit the time required to run the simulation.
3. Perform the simulation runs and epochs according to the evaluated policy and the learning strategy investigated;
4. Reuse the initial timing seeds to collect evaluation data for each policy (with learning rate and exploration parameter set to zero).
5. Evaluate whether the learned policy converged to an optimum, or achieved significant improvement over the random policy, using statistical analysis.
6. If the quality index of the results is larger than 0.1 (poor significance), increase the number of runs and repeat from step 3.
7. If the results show no significant improvement, nor convergence, then trial other parameter values such as different learning rate, epsilon, gamma (discount factor), lambda (eligibility trace decay), number of neural network layers, and number of epochs. Repeat from step 3.

Table 4.3 Experimental procedure.

Following this procedure, we obtain set of measurements for each CND scenario described. We present and explain these results in the next Chapter.

Chapter 5

Results Analysis

In this Chapter, we present the results of our experiments. We first present the results of the asset value algorithm introduced in **Chapter 3**, which assigns an importance scalar value to each asset of the CND environment. These values, which represent the business importance of each asset, are later used as damage scalars to compute risk values. We present some preliminary risk results for a simple case with an asset exposed to multiple vulnerabilities followed by an exploit. We finally present and discuss simulation runs results, measured in terms of evolution of risk values over time and its integral for each of the five scenarios described in **Chapter 4**.

5.1 Asset value results

We computed the contribution of each asset to the stated business needs using the asset valuation algorithm introduced in **Chapter 3**. This algorithm assigns values to assets based on the ratio of business-enabling network paths they support. Initially, only *User Services* assets have values. These values get percolated throughout the network, to other assets, including *User Services* themselves. In our simple CND environment, we found sixty-six paths through greedy, depth-first, search. The details of these dependency paths are exposed in Annex B. The resulting asset values are shown in Table 5.1 and Figure 5.1.

Asset name	**Asset Value**	**Asset initial value (*User Services*)**
Server-1	0.7	
Email-2	0.7	0.3
Browser-3	0.3	0.3
Router-4 (WWW)	0.3	
Email-5	0.7	0.4
Server-6	0.7	
DNS-7	0.3	
Router-8	0.72	
Router-9	1.0	
Router-10	0.72	
Router-11	0.88	

Table 5.1 Asset value results

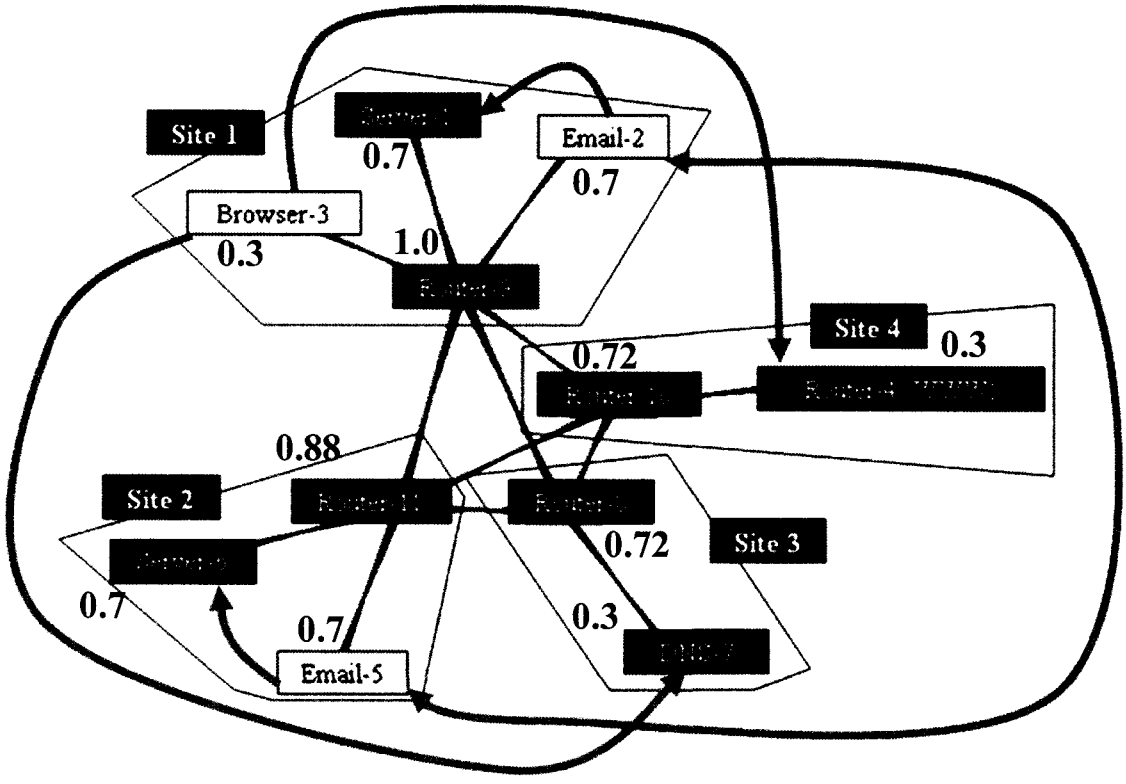

Figure 5.1 CND environment with resulting asset values.

These results can be empirically explained referring back to Figure 5.1. The asset value obtained for both email clients is the same at 0.7. Since both have a mutual functional dependency, they each inherit the value of the other, 0.3 + 0.4 = 0.7. Similarly, each email server has the same value as its client. The routers are more interesting to assess. The *router-9* is shown to have a value of 1.0, which means it is a single point of failure in our scenario. This asset is a site access point, and both browsing (0.3) and emails (0.3 + 0.4) loose essential communication links without it. In the case of *router-11*, the value of 0.88 can be explained by its support to both email clients (0.7) and to some of the paths enabling the browser to reach *DNS-7* and *router-4* (WWW gateway), for an additional 0.18. This result captures the idea that more than just the email clients are affected if this asset is severed from the network. It is also bounded by the sum of the values of affected *User Services* (≤ 1).

5.2 Risk Assessment results

A dynamic risk assessment algorithm is used to reward the learning agents and provide metrics to compare the overall risk between policies in given simulation scenarios. This algorithm was

introduced in **Chapter 3** and consists in the sum of the product of damages and probabilities, for all events and assets. Because assets are interdependent, risk results can be difficult to interpret. The graph of Figure 5.2 shows a simple case of how R(t) increases as multiple vulnerability events and an exploit event occurs at different time on a single asset (*email-5* from **Chapter 4**).

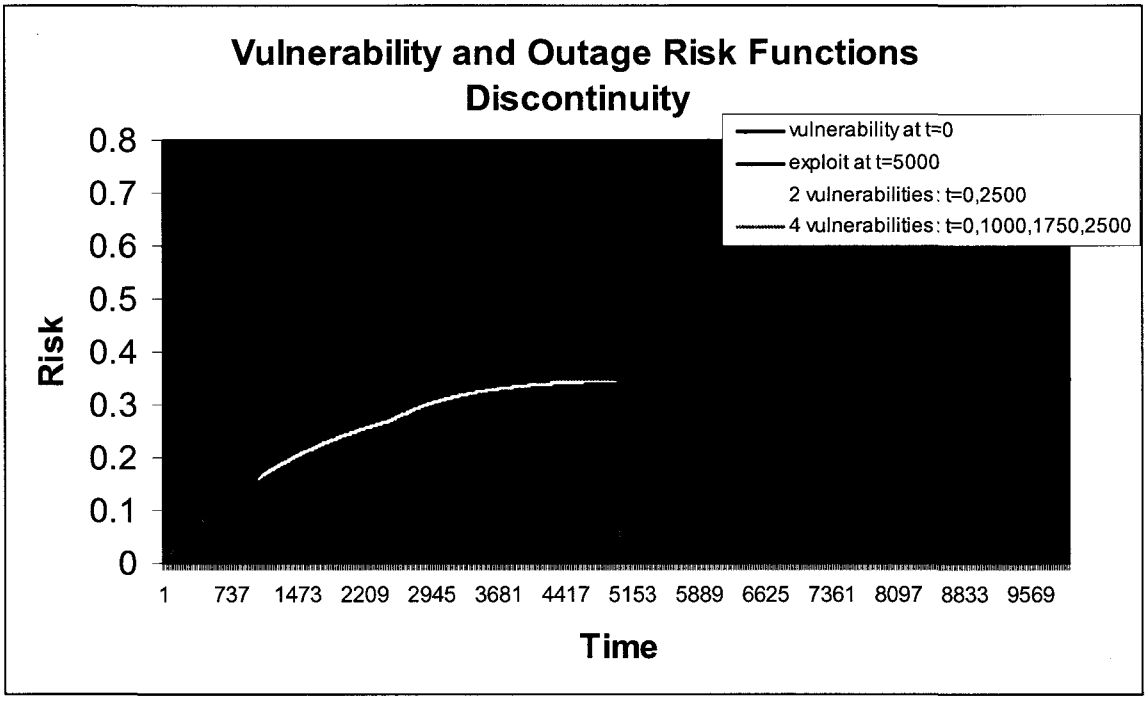

Figure 5.2 Evolution of the risk function R(t) with vulnerability and exploit events arrivals

We can observe that the risk slope increases with the arrival of additional vulnerabilities on the asset, supporting the idea of an increasing likelihood of exploit, as described in **Chapter 3**'s algorithm. The exploit scheduled at t=5000 causes an outage of the asset. Before this exploit occurs, the risk function is asymptotic to 0.35, that is, half of *email-5* value, as shown in Table 5.1. The discontinuity in **R(t)** is caused by a misalignment between the probability function **p(t)** and the damage function **d(t)**. This point marks the boundary between a proactive (potential damage) and a reactive (actual damage) decision. It captures the paradox that even a highly vulnerable asset cannot contribute to the risk, regardless how long it has been vulnerable for, as much as an asset of similar value which is actually out of service for a relatively short time. However, immediately after an outage occurs, **p(t)** = 1, while **d(t)** falls near zero due to the time sensitivity of the supported *User Services* (refer to variable ***b*** in **Chapter 3**). We solve this problem by programmatically preventing an asset risk contribution to decrease unless a mitigating action is actually implemented. The empirical choice of limiting to half of the asset value the risk of potentially damaging events (vulnerabilities) also captures the idea of risk aversion in existing CND decision making processes, as pointed out in **Chapter 1**.

5.3 Simulation Scenario results analysis

We conducted simulation runs for each of the scenarios introduced earlier using the methodology described in **Chapter 4**. For each scenario, the comparative results between the RL Neural Network policy, Table policy and the random policy are shown in Table 5.2. These results show statistically significant improvement of the RL policies over the random action selection policy. As we will discuss in **Chapter 6**, no further conclusions can be made with regard to the size of the improvement between policies. This is partly due to the fact that the actual optimal risk result for each scenario is not known. The scenarios and learning agents parameters used for each simulation can be found in Annex C.

Scenario	Random (avg risk)	RL Table policy (avg risk)	Improvement (avg risk)	RL Neural Network policy (avg risk)	Improvement (avg risk)
1. Fix 4 outages or wait.	31.73	16.38	**-15.35**	16.38	**-15.35**
2. Fix 11 outages	17.38	16.45	**-0.93**	13.42	**-3.96**
3. Patch 11 vulnerabilities	1.87	1.60	**-0.27**	1.77	**-0.10**
4. Patch 1 vulnerability or fix exploit	1.88	1.61	**-0.27**	1.81	**-0.07**
5. Patch, fix, isolate, or wait with continuous event arrivals	18933	18402	**-531**	18327	**-606**

Table 5.2 Simulation results for two reinforcement learning policies compared to the random policy.

In the simple case of four initially affected assets (scenario 1), reinforcement learning agents actually converged to a unique optimal risk value. We measured the convergence rate for each of the learning agents and the results are shown in Figure 5.3 for QConnectionist NN policy and Figure 5.4 for the Q-Learning Table policy.

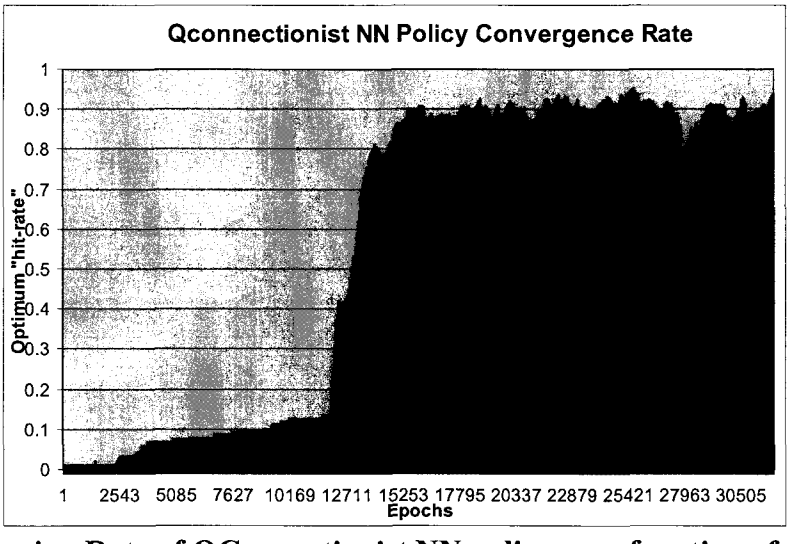

Figure 5.3 Learning Rate of QConnectionist NN policy, as a function of training epochs.

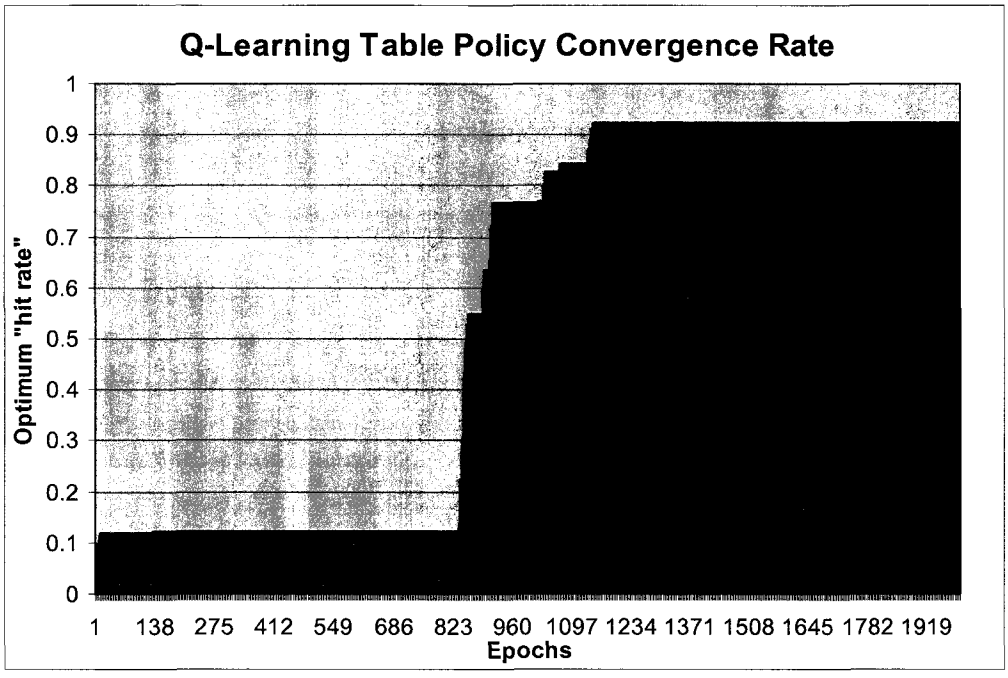

Figure 5.4 Learning rate of Q-Learning with table policy as a function of epochs.

Both the QConnectionist NN policy and the Table policy show near 90% hit rate of the optimal risk value. This is expected as the exploration rate (ε-Greedy) is set to 10%. The hit rate is computed using a sliding window technique. The jitter in the QConnectionist case is explained by the periodic mutations of 10% of the neural network's neurons to try to find a better optimum, and by the periodic roll back to the last best policy if no improvement is noted after 300 epochs. The sharpness in the convergence is caused by the high learning rate and the dependencies between assets.

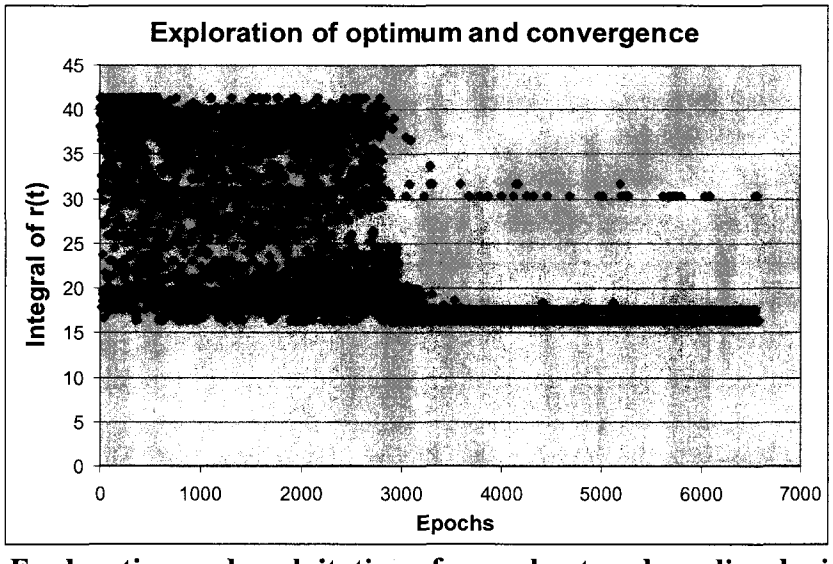

Figure 5.5 Exploration and exploitation of neural networks policy during training.

Before convergence, the agent explores the solution landscape and updates its target objective as new optima are found. When an invalid action is returned by the policy, a random valid action is selected. In Figure 5.5, we can observe the exploration and convergence pattern of the QConnectionist agent. We notice a trend at around $\int R(t) \approx 32$, which is the result for always selecting random actions (as shown in Table 5.3). In these cases, the agent returned no valid action every time. Any results >32 were caused by choosing "wait" actions. These choices were punished and became less frequent as training progressed. We can also observe a secondary periodic optimum after convergence at around $\int R(t) \approx 17$. This value corresponds to fixing assets in order of value as shown in Table 5.3. The upper bound of this graph is approximately $\int R(t) \approx 41$, which corresponds to taking no actions. In these cases, the agent decided to wait for the entire simulation period.

Policy	Integral of R(t)
Let risk grow	41.4
Fix random	31.73438
Fix highest	17.5375
Q-Connectionist	16.3825
Q-Learning Table	16.3825

Table 5.3 Final risk integral values converged to by each policy.

The $\int R(t)$ results for each test policy are shown in Table 5.3. Both Reinforcement Learning policies converged to the same optimum, which is better than the policy of fixing assets in order of value. This result is interesting and can be explained by the reuse of timing seeds for each epoch, and the *User Services* value chain. The list of affected assets in order of value for this scenario is: *router-8*, *email-5*, *DNS-7*, and *browser-3*. We know that *router-8* contributes to all three user services and that {*router-8, DNS-7, browser-3*} form a dependency chain with *browser-3* (value 0.3, as shown in Figure 5.1). In other words, all three assets must be fixed before *browser-3 User Service* is restored. The optimal policy found, given the actions timings of this simulation, is to repair this entire chain before fixing the *email-5* asset. We interpret this result as a local optimum specific to this timing sequence.

Unlike the NN policy, the resulting Table policy can be observed and interpreted. Table 5.5 below shows the optimum policy obtained. The state vector corresponds to the status of the four assets monitored ("0" for "OK", and "1" for "outage"). The "Best Action" column represents the action with the maximum Q-value for this state. Finally, the Q-value column represents the actual reward percolated to this state-action pair using the Q-learning algorithm described in **Chapter 3**. There are 5 possible actions (**n+1**). The action "0" means "Fix asset-3" and the action "1" means "Fix asset-5", and so on until action "4" which means "wait". Initial state at t_0

is {1,1,1,1} and the optimal action sequence to reach state {0,0,0,0} is fixing assets {8,3,7,5}. The "wait" action does not appear in the policy.

State Vector				Best Action	Q-Value
Asset3	Asset5	Asset7	Asset8		
0	0	0	0	3	0
1	0	0	0	0	9.01
0	1	0	0	1	0.909
1	1	0	0	0	0.243
0	0	1	0	0	0
1	0	1	0	2	2.6999976
0	1	1	0	2	2.6973002
1	1	1	0	0	0.6561046
0	0	0	1	0	0
1	0	0	1	3	5.14E-11
0	1	0	1	3	8.30E-07
1	1	0	1	3	0.0656116
0	0	1	1	1	0
1	0	1	1	3	1.23E-05
0	1	1	1	3	6.56E-06
1	1	1	1	3	0.27

Table 5.4 Q-Learning table policy example for fixing four (4) initial outages.

The fixing sequence of eleven outages (Scenario 2) is a problem with a much larger solution space (11! possible sequences). We conducted several simulations with the aim of converging to a single optimal policy, as for scenario 1, but without success. Although the learning agent can find through exploration optimal sequences leading to better risk results than fixing the highest asset first, it cannot converge to these optima. We therefore relied on statistical analysis for this scenario and the following ones.

To visualize some of the policies learned, we can replay the decisions made over a single simulation run, as shown in Figure 5.6. The associated risk integrals are presented in Figure 5.7. In this specific run, the agents learned a policy slightly inferior to fixing the highest asset value first. These results over a single run cannot be interpreted further as they have no statistical significance over the full sample space of timing sequences.

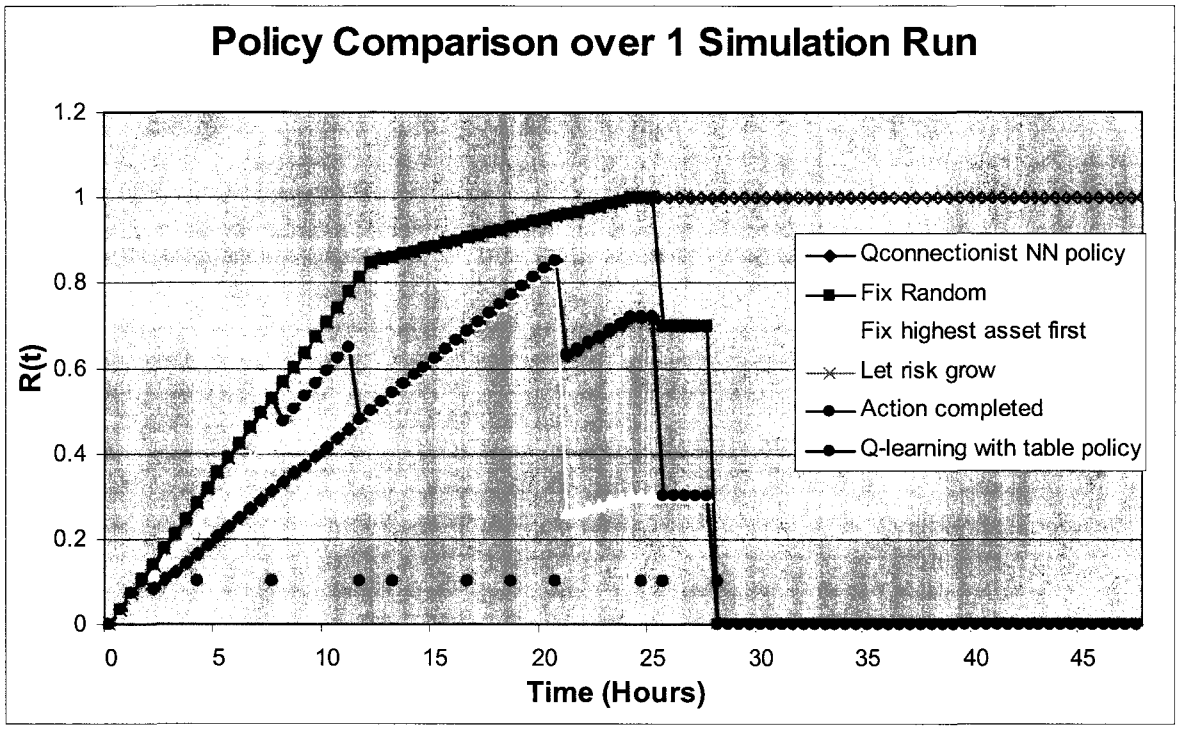

Figure 5.6 Policy comparison for fixing 11 outages.

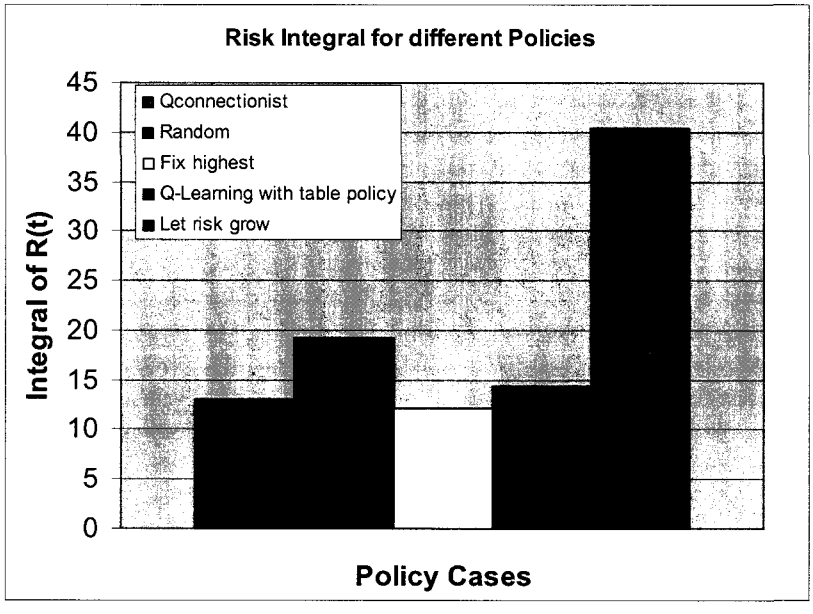

Figure 5.7 Policy comparison: risk integral for 1 run.

For the continuous simulation scenario (scenario 5), the training strategy is to reward agents for individual actions, independently of the overall risk integral. A single "epoch" is used, which means that the same timing sequence is never repeated. A learning episode, or sequence of

actions until the goal is reached, consists in every period with non-zero risk during the simulation.

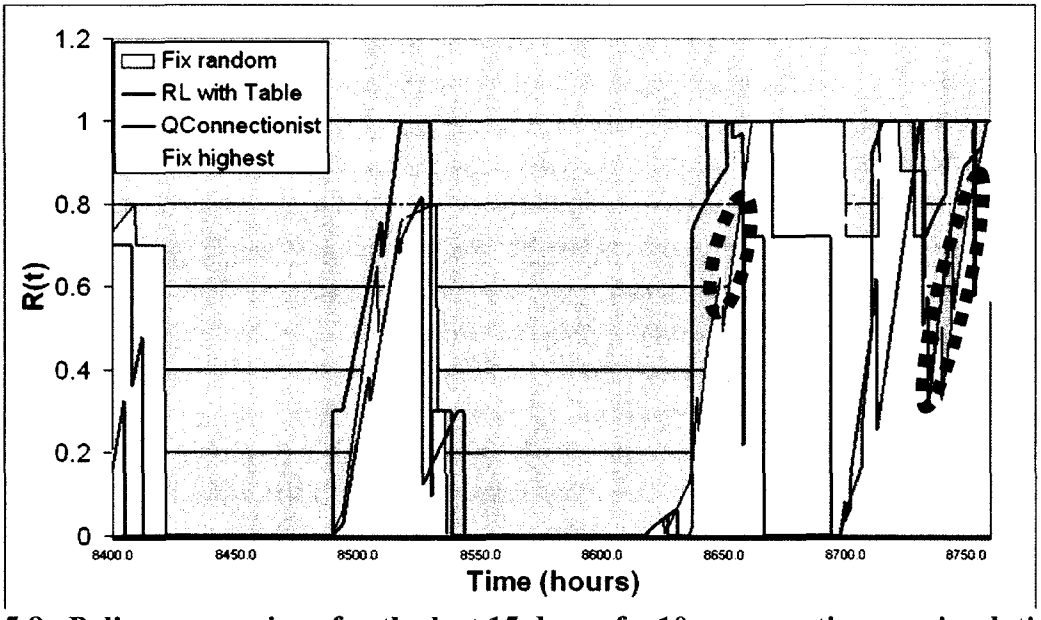

Figure 5.8 Policy comparison for the last 15 days of a 10-year continuous simulation run.

A fifteen-day period, out of a 10-year continuous simulation, is shown in Figure 5.8. The turquoise fill represents the random policy area under R(t). Although, the random policy shows the highest risk over the full simulation period, as shown in Figure 5.9, it achieves local optima in various situations, as shown by the red dashed ovals in the graph of Figure 5.8. For these two short intervals, the random policy actually achieves the lowest risk. We can also observe that risk episodes alternate, separated by valleys with zero risk. This is an indication that the resources available to respond to events are sufficient and that the system has reached a form of steady-state.

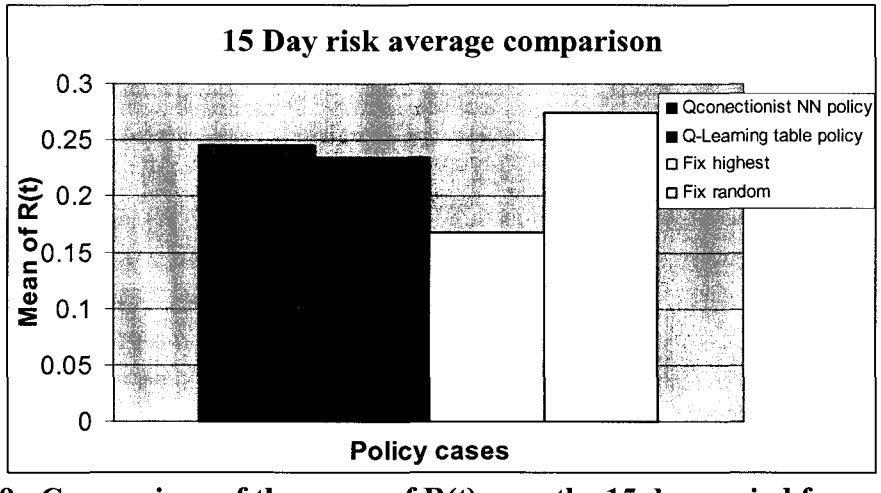

Figure 5.9 Comparison of the mean of R(t) over the 15 day period for each policy.

Additional continuous simulations were conducted to investigate the effect of the number of resources available on policy performance. As introduced in **Chapter 2**, a high number of resources should reduce overall risk exposure. We also argued that the limited availability of these resources should increase the need for a good CND policy. To validate that our risk metrics satisfy these conditions, we measured the integral of R(t), for two different policies (selecting affected assets randomly, or selecting the affected asset with the highest value first), with different number of NOC staff. The results are shown in Figure 5.10.

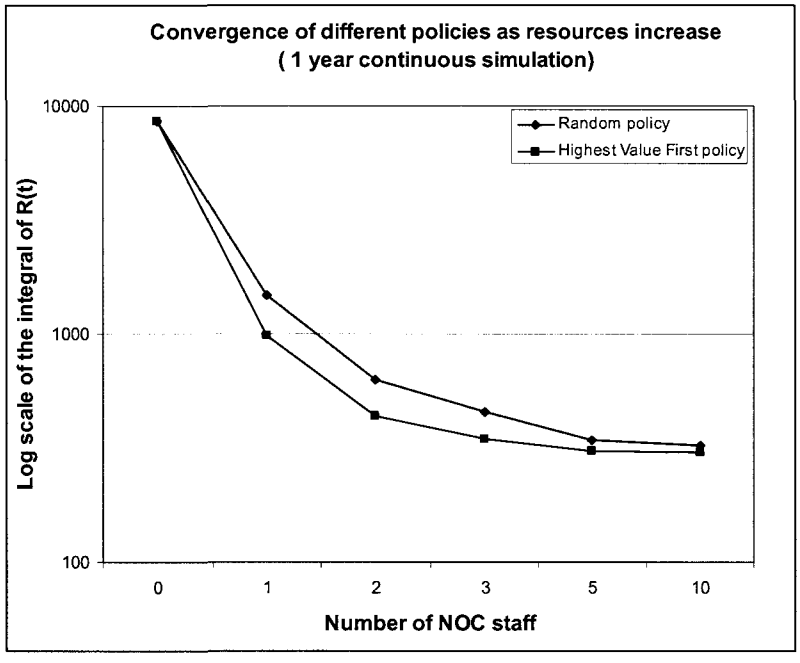

Figure 5.10 The effect of the number of NOC staff on the overall risk over a 1 year period.

The risk values shown in Figure 5.10 correspond to the average risk integral over 5 continuous simulation runs of 1 year. We can observe that the risk improvement of adding NOC staff decreases with respect to the logarithmic scale, as expected. The upper bound of the risk function corresponds to having no response capacity (zero NOC staff) and is given by:

$$8753 \leq 365 \text{ days} * 24\text{h} * 1.0 \text{ damage} = 8760.$$

Another observation is that, as the number of NOC staff increases, the difference between policies decreases (excluding the first point, since it is trivial that all policies will lead to the same risk if no action resource is available). This can be explained by the reduced need to prioritize responses. If there are no resource constraints, decision-making becomes trivial since all events can be immediately addressed as they arrive (no queue). This observation clearly supports the value of automation in CND as it is equivalent to increasing available resources.

The experimental results herein described and their meanings for our understanding of Autonomic CND are further discussed in the next Chapter.

Chapter 6

Discussion

Most of the policies learned by the RL agents did not represent global optima. Our results show that, following the NN policies and the table policies improved the overall risk over random action selection. However, the simple policy "fixing or patching the asset with the highest value first", without "isolate" or "wait" as action options, was superior in all scenarios but the first. This result may be explained by the characteristics of each tested scenario as well as the state model used, as we will now further discuss.

In scenario one (1), the size of the state and action space was much smaller than in other scenarios. This allowed the agent to fully explore the solution landscape and repeat positively reinforced action sequences sufficiently often so that each action of the optimal path had the highest Q-value for each relevant system state. In other scenarios, the agent may have followed an optimal path during exploration, but would not revisit this path sufficiently often to ensure each of its action's Q-value exceeded all other valid actions.

In scenario two (2) and three (3), the RL controller had to optimally align eleven (11) consecutive decisions before receiving a reward. Due to the dependencies between assets, the position in the path of every decision affects the final reward. This is fundamentally different from the maze problem introduced in **Chapter 3**. In the latter, regardless of the first decision, the last move before the goal always received the same reward. In the trialed CND scenarios, the last action may have lead to a penalty or a reward, depending on the entire decision sequence. As a result, the learning strategy of rewarding only at the end, once the risk is returned to null, lead to contradictory rewards. This behavior imposes that a higher number of epochs be used to converge to an optimum policy and that the reward received for the optimal path be large enough to compensate for conflicting cases.

Another important aspect of our experimentation is the use of timing seeds. In itself, the variation in the arrival times and in the action completion times can account for the RL policies failing to converge to an optimum. As we have shown in scenario one (1), a local optimum may exist for each individual timing seed. Changing these seeds during training created a moving target for some epochs by affecting the order of state change events, which in turn, affected the overall risk rewards. As a result, a reinforced decision path for one seed may have lead to a punishment for another.

Addressing the asset with the highest value first was generally a good policy. This is partly explained by the asset value being one of the main variables in the risk assessment algorithm. The asset value is also an extremely rich feature which accounts for all systemic relationships. However, this single-feature policy does not consider time as a decision factor. Hence, it will always address a recent event on a high value asset before an older event on a lower value asset, even though the risk contribution might be higher in the latter case. This sometimes leads to sub-optimal conditions, as shown in scenario 5's results. However, it also avoids worst situations whereby a long commitment to a low value asset causes the full damage of the higher value asset to realize. In other words, by committing to the highest value asset, the maximum risk is bounded by the value of the next highest value asset, regardless of the timing seeds. The case of exploits spreading represents an exception, as the decision taken influences future events and can therefore lead to higher risk than the next highest asset value. However, since our test environment is small, and since this only applies for product groups with more than one asset instance (like the five routers, two servers, two emails), this situation was relatively rare.

The policy addressing the highest asset value first was also helped by never choosing "wait" or "isolate" actions. As pointed out, the exploit spreading situation was rare, but was also the only case for which "isolation" actions may be justified. Furthermore, for both "wait" and "isolate", the controller would have benefited from clues as to how long this action has been applied for. From experience, we know that real NOC staffs often wait a few minutes, depending on the event, before taking any action, avoiding committing to transient events such as temporary bandwidth alarms or sensors' false positive which might self-correct. However, if the initial event's arrival time cannot be remembered (as captured in a ticketing tool for example), the "waiting" decision would perpetuate itself and lead to the event never being addressed. This situation is encountered in our system since the system state representation does not include the last action taken, or the last risk value. As a result, the controller cannot differentiate between the first time it decided to "wait" and the one-hundredth time, and therefore, this option may offer no obvious benefit.

The QConnectionist framework with the NN policy generally took more training epochs than Q-Learning with table policy, before showing meaningful improvement. This is partly caused by the neural network updating function and the number of hidden layers used, which further discount the reward through the sigmoid function. However, the table policy size was troublesome and even exceeded our Java virtual machine memory (2 GBits) for large scenarios such as the continuous case, and forced some workaround to allow completing the simulation. In the latter scenario, the policy size was over 184 millions entries.

The learning agents may have converged to a policy as good, or better than, "fixing the asset with the highest value first" if given more learning epochs. Although the timings space is

infinite, a longer simulation may have displayed a form of steady state behavior, with clear patterns for the learner to leverage. However, we could not observe this behavior conclusively in our experiments. The fact that the learning agents did not converge to a global optimum policy suggests that our simulation runs and training strategy may have been insufficient for such steady state behaviors to be observed and learned.

The results of the continuous simulation scenario show that all policies were, for short periods, optimal. This is due to the dynamics of the system which create a large number of transient behaviors. This observation supports our hypothesis that risk should be assessed over a time horizon (and that this horizon should be long enough). It may also indicate that policies can be specialized in a given set of situations, which would in turn support future experimentations with hybrid policy models.

Another observation is that the difference between policies decreases as more resources are available. This is due to the fact that every event can be served by NOC staff without delay in the queue, which makes decision making trivial. This supports the argument in favor of automation of CND actions, which is equivalent to having a near infinite supply of NOC staff, and therefore, may drastically simplify the controller's policy. These results also aligns with Moitra [55] statement that shortening response time have significant overall benefits in incident simulations.

Generally, we found result analysis of our experiments challenging due to the size of the data samples collected at run-time and the processing time of the different simulations. The sampling of the R(t) function proved to be a major processing load, and the table policies were difficult to hold in memory. Otherwise, the integration of the discrete event simulation environment, the dynamic risk assessment algorithm and the RL algorithms worked well and allowed for flexibility in the experimentation, whether for changing learning agents parameters, or for collecting system behavior traces during simulations.

The next Chapter summarizes this thesis' results and observations. It presents our conclusions, the limitations herein discussed and the potential areas for future work.

Chapter 7

Conclusion and Future Work

In this thesis, we demonstrated a simulated autonomic CND system which leverages dynamic risk assessment and reinforcement learning to improve response policies. We showed that policy iteration for CND is possible using the proposed framework. We further showed that a response policy using a table or a function approximator can reduce the risk over an explicit period of time, for a limited CND environment. We showed that in a simple scenario, these policies converge to the same optimal policy. However, our results suggest that reinforcement learning agents could not converge to a globally optimum policy in more complex scenarios. An empiric policy which selects actions based on affected assets value was shown to be better in these cases. These results confirm Bursztein's observations in [15] and strongly support the use of a pre-computed asset value to prioritize network event responses. It also suggests that the use of reinforcement learning should focus on finding locally optimal policies, for very specific scenarios, rather than attempt to learn a single globally optimum CND policy.

Limitations

Our work suffers from a number of limitations. We did not conduct experiments over a large IT network. This is common in research of this type because of the difficulty to interpret results in complex environments. Nevertheless, real size network scenarios would be an essential validation step for the autonomic CND concept. This thesis also only reports on experiments with generic scenarios, as its intent was to investigate the potential, rather than finding an actual optimal approach. Focussing on specific scenarios may have revealed more applicable results. Another limitation, which should be pointed out, is that the simulated CND environment and risk algorithms did not include explicitly ***confidentiality*** and ***integrity*** as part of the potential damages. These represent additional layers of logical relationships between assets and *user services* which would have significantly increased the complexity of the environment. Namely, modelling *files* as assets, and *read-write permissions* as relationships would have been required to account for these security metrics. These additions could however be integrated into the models used in this thesis.

Future Work

Autonomic CND is a new area of research which this thesis only explores the surface of. Many areas require more work, as can be deducted from the limitations listed above. Specifically, future work may include conducting in-depth research of some of the key topics covered in this thesis. These include the dynamic risk assessment algorithm, the CND state representation and the policies themselves.

The dynamic risk assessment we proposed was developed based on empirical observations which should be validated against experts' opinion and possibly other risk valuation techniques. Also, the underlying CND graph model we proposed has yet to be used in a realistic environment. We would expect that an exhaustive review of IT management processes be required to maintain such a model operationally.

Another future work area concerns the CND environment state representation. The CND states used for learning policies were translated into a simple vector with each element representing an asset status. In a large environment, this vector would grow to several thousands entries. As we showed, the state space drastically impairs the learner's ability to converge. To improve this situation, the CND state representation may be able to leverage text mining techniques. One possible avenue to explore would be metaphorically presenting the CND graph model as a semantic network. Interdependencies between assets would be analogous to semantic relationships between words in a text. Since text mining has shown that texts classification tasks did not require word relationship information to be successful, we believe there is much potential for research in looking at IT network states modeled as a *bag-of-words*. This representation could enable feature ranking (for which our asset value algorithm could be used), filtering, selection and clustering techniques to improve convergence. It is not clear how these techniques would be applied to time-series like the chain of CND actions from a policy.

Finally, our work showed that policies themselves would not scale to large IT environment. Therefore, another possible area of future work would be to investigate distributed policy models, including using Collaborative Reinforcement Learning, in larger simulated and lab environments. This approach may be able to leverage the hierarchical nature of IT networks to derive efficient localized policies. A concern with this approach however would be stability of the whole system as reconfiguration actions take place simultaneously in many network regions.

Bibliography

[1] Alhazmi, Malaiya, "Quantitative Vulnerability Assessment of Systems Software", Reliability and Maintainability Symposium, 2005.

[2] Australian Communications-Electronic Security Instructions, 2004.

[3] Barto, Crites, Improving Elevator Performance Using Reinforcement Learning, Adances in Neural Information Processing Systems 8, 1996.

[4] Bass T, "Intrusion detection system and multisensors data fusion", communication of the ACM, April 2000, vol 43, no 4.

[5] Beaudoin, "Severity Classification using Network Events and Operations Dependencies", University of Ottawa CSI5384 project,April 06.

[6] Beaudoin, Asset Valuation Technique for Network Management and Security, IEEE ICDM Workshop proceedings, Dec 06.

[7] Beinhocker, "The Origin of Wealth", Harvard Business School Press, 2006.

[8] Benjamin, Pal, Webber, Atighetchi, Ruber, Automating Cyber-Defense Management, ACM Workshop on Recent Advances in Intrusion Tolerant Systems, 2008.

[9] Bertels, Nami, "Survey of Autonomic Computing Systems", Proceedings of the Third International Conference on Autonomic and Autonomous Systems, 2007.

[10] Birta, Arbez, "Foundation on Modeling and Simulation", University of Ottawa, 2006.

[11] Bowles, "Microeconomics: Behavior, Institutions and Evolution", Russel Sage foundation, 2004.

[12] Boyan, Littman, Packet Routing in Dynamically Changing Networks: an RL approach, Advances in Neural Information Processing Systems, volume 6, pages 671--678. Morgan Kaufmann, San Francisco CA, 1993

[13] Brigham, Kahl, Rentz, Gapenski, "Canadian Financial Management", 4th edition, Harcourt Brace and Company Canada ltd, 1994.

[14] Burns, Cheng, Gurung, Rjagopalan, Rao, Rosenbluth, Surendran, Martin, Automatic Management of Netwotk Security Policy, Telcordia DISCEX 2001.

[15] Bursztein, Using Strategy Objectives for Network Security Analysis, INSCRYPT 2008

[16] Cao, From Perturbation Analysis to Markov Decision Processes and Reinforcement Learning, DEDS: Theory and Application 2003.

[17] Chairman of the Joint Chiefs of Staff Instruction, "Information Assurance and Computer Network Defense", US DoD, 2004.

[18] Chan H, Kwok T, "An Autonomic Problem Determination and Remediation Agent for Ambiguous Situations Based on Singular Value Decomposition Technique", IEEE IAT International Conference proceeding, Dec 06.

[19] Colajanni, Andrealini, Lancellotti, "Open Issues in Self-Inspection and Self-Decision Mechanisms for Supporting Complex and Heterogeneous Information Systems", SELF-STAR 2004.

[20] Dadone, Vanlandingham, Maione, Modeling and Control of Discrete Event Dynamic Systems: A Simulator-Based Reinforcement-Learning Paradigm, International Journal of Intelligent Control and Systems 1998.

[21] Dailey, Harn, Lin, "ITS Data Fusion", Washington State Transportation Centre, 1996

[22] Dickson, "An Operational Framework for Federated Network Management", EDUCAUSE SPC 2007.

[23] Diettrerich, Flann, "Explanation-Based Learning and Reinforcement Learning: A unified View", Kluwer Academic Publisher, 1997.

[24] Dobson, Denazis, Fenandez, Gaiti, Gelenbe, Massacci, Nixon, Saffre, Schmidt, Zabonelli, A survey of Autonomic Communications, ACM Autonomous and Adaptive Systems, Vol. 1, No. 2, 2006.

[25] Dobson, Massacci, "A Survey of Autonomic Communications", Transactions on Autonomous and Adaptive Systems, 2006.

[26] Dowling, Cahill, "Self-Managed Decentralized Systems using k-components and Collaborative Reinforcement Learning", workshop on Self-managed systems, 2004.

[27] Duncan, Luce, Raiffa, "Games and Decisions: Introduction and Critical Survey", John Wiley & Sons, 1957

[28] Feiertag, Rho, Redmond, Achieving Coordination and Control of Cyber Defense Through Dynamic Policy, http://securitycore.cougaar.org/docs/PolicySystem.pdf.

[29] Fox, Kiciman, Patterson, Katz, Jordan, Stoica,"Statistical Monitoring + Predictable Recovery", SELF-STAR, 2004.

[30] Francesco De Comité, "PIQLE, A platform for Implementation of Q-Learning Experiments", NIPS 2005.

[31] Garlan, Schmerl, "Model-based Adaptation for Self-Healing Systems", Proceedings of the first workshop on Self-healing systems, 2002.

[32] Gigerenzer, Goldstein, "Reasoning the Fast and Frugal Way: Models of Bounded Rationality", American Psychological Association,1996

[33] Gotaishi, "Business Continuity Planning beyond ISO17799", SANS Institute 2004.

[34] Gula, "Network Security Implications of Visible Ops", Tenable inc, 2007.

[35] Hariri, Qu, Dharmagadda, Ramkishore, Raghavendra, "Impact Analysis of Faults and Attacks in Large-Scale Networks", IEEE Security and Privacy, 2003.

[36] Harmon M, Harmon S, "Reinforcement Learning: A Tutorial", http://citeseer.ist.psu.edu/harmon96reinforcement.html, 1996.

[37] Haverkort, "Model-Based Self-Configuration for Quality-of-Service!", SELF-STAR 2004.

[38] Hensel, "A Scalable Approach to Automated Service Dependency Modeling in Heterogeneous Environments", University of Munich,1999

[39] Holcombe, "US Homeland Security Market Forecast:$140Billions", Homeland Security Weekly, 5th Jan 2007.

[40] Hummel, Kosub, "Acyclic Type-of-Relationship Problems on the Internet: An Experimental Analysis", Internet Measurement Conference 2007.

[41] IBM inc, "The Tivoli software implementation of autonomic computing guidelines", IBM, 2002.

[42] ITIL Best Practices, Office of Government Commerce, 2000, www.tso.uk/ITIL

[43] Jin, Tsai, "Temporal Network Analysis for Predictive Routing Table Optimization", Berkeley, 2001.

[44] Johansson, "Information Acquisition in Data Fusion Systems", Swedish Defence Research Agency (FOI), 2003

[45] Knight, McIntyre, An Operational Framework for Battle in Network Space, 10th International Command and Control Research and Technology Symposium, 2006

[46] Kotenko, Multi-agent Modelling and Simulation of Cyber-Attacks and Cyber-Defense for Homeland Security, IEEE International Workshop of Intelligent Data Acquisition and Advanced Computing Systems, 2007

[47] Kumar, Miikkulaimen, "Confidence Based Dual Reinforcement Q-Routing: An adaptive online network routing algorithm", Proceedings of the Artificial Neural Networks in Engineering Conference, 1998.

[48] Kuzmin, "Connectionist Q-Learning in Robot Control Task", Riga Technical University, 2002.

[49] Lambert (AS), E. Bossé (Can), R. Breton (Can), R.Rousseau (Can), J.R. Howes (UK), M.L. Hinman (US),J. Karakowski (US), M. Owen (US), F. White (US),"Information Fusion Definitions, Concept and Models for Coalition Situation Awareness", TTCP C31 Group, TR-C31-AG2-1-2004, April 2004.

[50] Lefebvre, Grégoire,Froh,Beaudoin, "Computer Network Defence Situation Awareness Information Requirements", MILCOM 2006.

[51] Lemos, "Counting the cost of Slammer", ZDNet Australia, 2003.

[52] Liu, Parashar, "Accord: A Programming Framework for Autonomic Applications", ICAC 2004.

[53] Martin-Flatin, "Distributed Event Correlation and Self-Managed Systems", CERN, 2004.

[54] Mitchell T.M, Machine Learning, McGraw-Hill, 1997

[55] Moitra, Konda, "The Survivability of Network Systems: An Empirical Analysis", CMU SEI, 2000.

[56] Murata, "Biologically Inspired Communication Network Control", International Workshop onSelf-* Properties in Complex Information Systems, 2004.

[57] NIST, "CND Data Strategy and Security Configuration Management", SCAP 2008.

[58] Nowicki, Squillante, Wah Wu, " Fundamentals of Decentralized Optimization in Autonomic Systems", IBM, SELF-STAR 2004.

[59] O'Hare, Noel, Prole, "A Graph-Theoretic Visualization Approach to Network Risk Analysis", VizSec 2008.

[60] Ou, Govindavajhala, Appel, "MulVAL: A Logic-based Network Security Analyzer", Princeton University, 2005

[61] Ou, Rajagopalan, Rakshit, Sakthivelmurugan, An Empirical Approach to Modeling Uncertainty in Intrusion Analysis, SNIPS software
http://people.cis.ksu.edu/~xou/argus/software/snips/

[62] Precup, Sutton, Dasgupta, "Off-Policy Temporal-Difference Learning with Function Approximation", 18th Conference on Machine Learning, 2001.

[63] Rieke, Roland, "Tool based formal Modelling, Analysis and Visualisation of Entreprise Network Vulnerabilities utilizing Attack Graph Exploration", EICAR 2004.

[64] Ross, "Introduction to Probability Models", 9th edition, Elsevier inc, 2007.

[65] Ryan, "iWar: A new threat, its convenience – and our increasing vulnerability", NATO review, 2007.

[66] Sawilla, Ou, "Identifying Critical Attack Assets in Dependency Attack Graphs", DRDC TM, 2008.

[67] Stephenson, " A Formal Model for Information Risk Analysis Using Colored Petri Nets", Michigan University, 2004.

[68] Sutton, "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming", 7th Conference on Machine Learning, 1990.

[69] Sutton, Barto, Reinforcement Learning: An Introduction, MIT press, www.cs.ualberta.ca/%7Esutton/book/ebook, 1998

[70] Sutton, McAllester, Singh, Mansour, Policy Gradient Methods for Reinforcement Learning with Function Approximation, Advances in Neural Information Processing Systems 12, 2000.

[71] Taylor, Tofts, "Self Managed Systems – A Control Theory Perspective", HP technical report, 2004.

[72] Tesauro, "Reinforcement Learning in Autonomic Computing", IBM T.J. Watson Research Center, IEEE 2007.

[73] Tesauro, Jong, Das, Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation", IBM Research Report, 2006.

[74] United States Department of Defense, "DoD Guide for Achieving Reliability, Availability, and Maintainability", DoD 2005.

[75] Wagner, Rowe, Rickards, Hites, Fischer, Bielec, Gonzales, Gonick, Wheeler, Jackson, Voss, Hogue, Katz, "The Organization of the Organisation: CIO's view on the Role of Central IT", EDUCAUSE 2007.

[76] Wang, "Computer Genomics: Towards Self-Change and Configuration Management", Microsoft Research, 2004.

[77] Wang, Singhal, Jajodia, "Toward Measuring Network Security Using Attack Graphs", Conference Proceedings on Computer and Communications Security, 2007.

[78] Watt (LCol, USAF), "Self-Inflicted System Malfunctions Threaten Information Assurance", SIGNAL 1999.

[79] Witten, Frank, "Data Mining", 2nd edition, Elsevier, 2005

[80] Wu, "Sensor Data Fusion for Context-Aware Computing Using Dempster-Shafer Theory", PhD Thesis, Robotics Institute Carnegie Mellon University, 2003

[81] www.psepc-sppcc.gc.ca/prg/em/ccirc/abo-en.asp

[82] www.wikipedia.org {Bellman,Data fusion,Dempster-Shafer,Kalman Fiters,Q-learning,Reinforcement Learning,Sensor Fusion,Earned Value}

[83] Zhuge, Holtz, Song, Guo, Han, Zou, "Studying Malicious Website and the Underground Economy of the Chinese Web", Reihe Informatik University of Mannheim, Germany, 2007.

Annex A

The CND Environment Simulation Conceptual Model

The following section describes the CND simulator conceptual model. It follows the framework described in ABCMod [10] for discrete event dynamic systems simulation. The java code implementation of this model is included in this thesis experimental implementation.

Constants and Parameters

*All times in hours, unless explicitly given.

Constants and Parameters		
Name	**Role**	**Value**
T0	Left boundary of Observation Interval	0(clock time)
Tf	Right boundary of Observation Interval	Steady state study required. Various simulation runs. Explicit condition for some at risk=0 (and t>0).
All_graph_Assets[i]	Collection of CND environment assets, which include user service assets.	.size = 11 Collection of assets
NumAssetForStates	For some scenarios, reduces the state space by only allowing this number of concurrent assets to be affected.	1-11
AssetsForStates[i]	For some scenarios, specifies which assets can be affected.	.size = 1-11 Collection of assets
NumAssetsStates	Number of states assets can be in. OK, outage, vulnerable, exploited.	2-4
Product[i]	For vulnerabilities, list of the unique products in the AssetForStates collection	.size = 1-11 Collection of asset types
PhysicalGraph	Collection of vertices and edges where vertices are assets and edges are communication paths.	JgraphT object (collection of assets and edges)
LogicalGraph	Collection of vertices and edges. May be composed of disconnected sub-graph. Represents logical dependencies between service providing assets.	JgraphT object (collection of assets and edges)
NumNocStaff	The number of personnel available to fix, isolate, patch, wait	1-3
NumAction	Action set available to the policy and NOC staff	1-4
OutageMean	Mean interarrival time between outages, follow Exp. Dist.	0.00036*NumAssetsForStates =0.00391
VulnMean	Mean interarrival time between vulnerabilities, follow Exp. Dist.	0.0134
ExploitMean	Mean delay between the release of a vulnerability and an exploit attempt.	0.0093
ExploitSigma	Factor of exploit risk increase for a vulnerability, according to $1-e^{-\Delta t/\gamma}$.	168 (one week)
FixingMean	Average time to fix an outage or an exploit.	2
WaitTime	Default wait time when ignoring existing	0.1 (6 min)

	events.	
FixMaxT	Maximum time to fix an asset (ex: total rebuild)	24
FixMinT	Minimum time to fix an asset (ex: reboot)	0.5
PatchMaxT	Maximum time to patch (ex: testing + manual deploy)	48
PatchMinT	Minimum time to patch (ex: automated minor update)	0.5
Risk	Double, represent current risk state of the system	0..1
Qconn_Agent	Neural network policy implementation	Qconn Obj
RLNetSim	Table policy implementation	RLNet Obj

Internal Data Modules

Internal Data Models		
Name	**Description**	**Data Model**
fixT	Time to fix an asset	Uniform(FixMinT, FixMaxT)
PatchT	Time to patch an asset	Uniform(PatchMinT,PatchMaxT)
IsolateT	Time to isolate an asset	Uniform(IsolateMinT,IsolateMaxT)
AssetID	Next asset affected by events	Uniform(1, all_Graph_Asset.size)
ProductID	Next product affected by new vulnerability	Uniform(1, Product.size)
ActionID	Next action selected when exploring (random selection)	Uniform(1, numAction)
AffectedAssetID	Next asset to be fixed from the list of affected assets.	Uniform (1, affectedAsset.size)

User specified Procedure Module	
SelectAction()	Calls policy and retrieve highest Q-value s,a pair for this state (greedy, softmax, EM-Boltzman)
updatePolicy()	Propagate error/reward to the eligibility traces and neurons using Q-value estimation formulas.
RiskCalculator()	Uses the Affected Assets set to compute the present level of risk.
getState()	Iterates over all graph assets to fill the state vector based on assets and resources status.

Consumer Entity

AssetsForState [i]	
This consumer entity class represents network assets susceptible to be affected by various events.	
Attributes	(Asset,N)
N	Size equal to numAssetsForState
Clock	Last state change time
vulnerabilityID_set	Collection of vulnerability IDs
Status	0=OK, 1=outage,2=vulnerability, 3=exploit
Value	Double, computed at model creation
Name	Unique name
Type	Type, also referred to as product.
PathID_set	Collection of paths supported by the asset
ZoneBorder	Collection of assets protecting the asset's zone.
isVulnerable	Boolean, true if vulnerabilityID_set.size>1.
isUserService	Boolean, true if asset is a user service
US_value	Double, value of the user service, also known as operational

	requirement.

Resource Entity

NOCStaff[i]	
This resource entity represents the NOC staff tasked to fix, patch, isolate or wait in case of network events.	
Attributes	(SKUSet, Status, Load, startIdle, TotalIdle)
Asset	Asset object being actioned.
Busy	Boolean: true = busy, false = Idle
Init_State	System state at NOC staff assignment time.
StartIdle	A time stamp used to determine waiting times.
TotalIdle	Accumulated waiting time

Aggregate Entities

AffectedAssets [i]	
This group represents all assets with a status $\neq 0$. The state of this group drives the risk calculation and represents an absorbing state when size =0, for some scenarios.	
Attibutes	(List, N)
List	A list of asset objects with state $\neq 0$. Discipline: varies. Random, highest value, FIFO are considered.
N	The number of entries in List. Bounded by numAssetsForState.

Inputs

Input				
Input Variable	**Description**	**Data Models**		**Action Sequence**
		Domain Sequence	Range Sequence	
Outage(t)	The occurrence of an outage on an Asset.	Affected Asset i = uniform(1,AffAssets.size) Inter-arrival = Exp(OutageMean)	Equal 1. Asset determined by newOutage()	newOutage()
Vulnerability(t)	The occurrence of a vulnerability on asset type i.	Vulnerable asset type = uniform(1, assetType.size) Inter-arrival = Exp(VulnMean)	Equal 1. leads to vulnerability instances and exploits.	NewVuln()
Exploit(t)	The occurrence of an exploit on a vulnerable asset.	Generated after newVuln() Arrival = vulnerability(t) + Exp(exploitMean)	Equal 1.leads to additional exploits.	newExploit(vulnerability.asset)
Sample(t)	The occurrence of a sampling of the risk function	Inter-arrival = 10 min	Risk value from RiskCalculator	heartbeat()

Action Sequence

Action Sequence: newVuln()

Precondition	t= Vulnerability(t)
Event	For each asset, if asset.type = vulnerability.product, Asset.isvulnerable =true; NewExploit(asset)

Action Sequence: newExploit(vulnerability.asset)	
Precondition	t= Exploit(t)
Event	If unprotected_pathExists from www.zone to asset.zone Asset.status = exploited For each asset, if asset.type = vulnerability.product, and unprotected_pathExists from vulnerability.asset.zone to asset.zone, then NewExploit(asset)

Action Sequence: newOutage()	
Precondition	t= Outage(t)
Event	Pick random asset from the AssetForState list until status = 0 (OK), If (numAssetForState < affectedAsset.size), in the case of state limiting scenarios. Asset.status = 1 (outage) Asset.assigned = false.

Action Sequence: heartBeat()	
Precondition	t= Sample(t)
Event	PHI[risk] = riskCalculator, clock

Outputs

Sample Set	
Name	**Description**
PHI[risk]	Risk values collected.

Derived Scalar Output Value			
Name	Description	Output Set Name	Operator
Risk Integral	The integral under the curve of risk values collected,	TotalRisk	Integral (rectangular bins approximation)

Model Behaviour

Summary of Activities	
Action Sequence	
Outage(t),Vulnerability(t),Exploit(t)	Affects the status of the AssetsForState goup.
Activities	
Fix()	Change an asset status from 1 back to 0.
Patch()	Change an asset status from 2 back to 0.
Isolate()	Change an asset status from 2 or 3 back to 0.
Wait()	Increment system clock only.
HeartBeat()	Update PHI[risk] sample set

Initialize
For each NOCstaff[i]

.busy<----------False

For each AssetForState[i], .status = 0 and .assigned = false.
AffectedAssets = empty set
Fel (future event list) = 0
Risk = 0

Policy = set to random values;

Activity : Fix	
Precondition	AffectedAsset.size > 0 NOCStaff[i].busy = false Case Random:Asset[DM.AffectedAsset].status = 1 Case Policy: Asset[selectAction()].status=1
Event	NOCStaff[i].busy = true Asset.assigned = true Asset.Event.clock = clock NOCStaff[i].Asset = Asset NOCStaff[i].lastState = getState()
Duration	DM.FixT
Event	tempState = getState() NOCStaff[i].busy = false Asset.assigned = false If Asset.vulnerabilityID_set > 0, .status = 2, else, .status =0. AffectedAsset.Remove(Asset) TempRisk = RiskCalculator(clock, AffectedAssets) Switch[Case continuous reward: Reward =PHI[risk].get(last-1) - TempRisk Case goal reward only If AffectedAsset.size = 0, Reward = 10, 0 otherwise] UpdatePolicy(learner, tempState, getState(), Fix, reward) lastState = getState

Activity : Patch	
Precondition	AffectedAsset.size > 0 NOCStaff[i].busy = false Case Random:Asset[DM.AffectedAsset].status = 2 OR . vulnerabilityID_set.size >0; Case Policy: Asset[selectAction()].status=2 OR . vulnerabilityID_set.size >0;
Event	NOCStaff[i].busy = true Asset.assigned = true Asset.Event.clock = clock NOCStaff[i].Asset = Asset NOCStaff[i].lastState = getState()
Duration	DM.PatchT
Event	tempState = getState() NOCStaff[i].busy = false Asset.assigned = false Asset.isVulnerable = false Asset.vulnerabilityID_set.Remove(.Event.vulnerabilityID)

	If Asset.vulnerabilityID_set = 0, .status = 0. AffectedAsset.Remove(Asset) TempRisk = RiskCalculator(clock, AffectedAssets) Switch[Case continuous reward: Reward =PHI[risk].get(last-1) - TempRisk Case goal reward only If AffectedAsset.size = 0, Reward = 10, 0 otherwise] UpdatePolicy(learner, tempState, getState(), Fix, reward) lastState = getState

Activity : Isolate	
Precondition	AffectedAsset.size > 0 NOCStaff[i].busy = false Case Random:Asset[DM.AffectedAsset].status = 2 or 3 OR . vulnerabilityID_set.size >0; Case Policy: Asset[selectAction()].status=2 or 3 OR . vulnerabilityID_set.size >0;
Event	NOCStaff[i].busy = true Asset.assigned = true Asset.Event.clock = clock NOCStaff[i].Asset = Asset NOCStaff[i].lastState = getState()
Duration	DM.IsolateT
Event	tempState = getState() NOCStaff[i].busy = false Asset.assigned = false Asset.ZoneBorder.status = 1 AffectedAssets.add(Asset.ZoneBorder) Asset.status = 3 TempRisk = RiskCalculator(clock, AffectedAssets) Switch[Case continuous reward: Reward =PHI[risk].get(last-1) - TempRisk Case goal reward only If AffectedAsset.size = 0, Reward = 10, 0 otherwise] UpdatePolicy(learner, tempState, getState(), Fix, reward)

Activity : Wait	
Precondition	AffectedAsset.size > 0 NOCStaff[i].busy = false Case Random:Asset[DM.AffectedAsset] = null Case Policy: Asset[selectAction()] = null
Event	NOCStaff[i].busy = true NOCStaff[i].lastState = getState()
Duration	WaitT
Event	tempState = getState() NOCStaff[i].busy = false

	tempRisk = RiskCalculator(clock, AffectedAssets) Switch[Case continuous reward: Reward =PHI[risk].get(last-1) - TempRisk Case goal reward only If AffectedAsset.size = 0, Reward = 10, 0 otherwise] UpdatePolicy(learner, tempState, getState(), Fix, reward) lastState = getState

Annex B

Asset Value Algorithm sub-results

There are 66 paths (generated using pseudo-code described in **Chapter 3**) supporting the business needs in the 11-asset scenario described in **Chapter 4**. The following list shows how each asset participates in a number of paths (PathID). Every time an asset value is updated for a given User Service, the pathID supported by this asset is added to the source and User Service. This allows for traceability. It will also enable risk computation when many assets are affected simultaneously. This risk calculation is described in the next section.

Paths and Asset value vector:

- PathID for asset 0=[44, 22, 0]
- User service value vector for asset 0 = [0.4, 0.0, 0.3]

- PathID for asset 1=[51, 30, 8, 23, 31, 48, 7, 54, 49, 22, 32, 53, 9, 6, 1, 29, 52, 24, 47, 4, 26, 50, 3, 27, 44, 45, 2, 28, 46, 25, 10, 5, 0]
- User service value vector for asset 1 = [0.4, 0.0, 0.3]

- PathID for asset 2=[62, 15, 40, 43, 16, 35, 59, 34, 41, 65, 21, 60, 61, 14, 42, 36, 19, 58, 63, 18, 39, 12, 64, 17, 13, 38, 20, 57, 37, 56]
- User service value vector for asset 2 = [0.0, 0.3, 0.0]

- PathID for asset 3=[15, 36, 58, 16, 35, 12, 59, 34, 13, 38, 60, 57, 14, 37, 56]
- User service value vector for asset 3 = [0.0, 0.3, 0.0]

- PathID for asset 4=[51, 30, 8, 23, 31, 48, 7, 54, 49, 32, 53, 9, 33, 6, 1, 29, 52, 24, 47, 4, 55, 26, 11, 50, 3, 27, 45, 2, 28, 46, 25, 10, 5]
- User service value vector for asset 4 = [0.4, 0.0, 0.3]

- PathID for asset 5=[55, 33, 11]
- User service value vector for asset 5 = [0.4, 0.0, 0.3]

- PathID for asset 6=[62, 19, 40, 43, 18, 63, 39, 64, 17, 41, 65, 21, 61, 20, 42]
- User service value vector for asset 6 = [0.0, 0.3, 0.0]

- PathID for asset 7=
- [62, 15, 51, 40, 43, 31, 48, 35, 7, 59, 41, 53, 9, 65, 21, 6, 61, 29, 14, 42, 24, 47, 4, 19, 36, 58, 26, 50, 63, 18, 3, 39, 64, 17, 2, 13, 28, 57, 46, 20, 25, 37]
- User service value vector for asset 7 = [0.24, 0.3, 0.18]

- PathID for asset 8=[62, 15, 30, 43, 16, 31, 48, 59, 32, 34, 41, 60, 61, 1, 29, 14, 47, 58, 63, 18, 3, 12, 27, 44, 64, 17, 2, 13, 28, 38, 57, 37, 51, 8, 23, 40, 35, 7, 54, 49, 22, 53, 9, 65, 21, 6, 52, 24, 42, 4, 19, 36, 26, 50, 39, 45, 20, 46, 25, 56, 10, 5, 0]
- User service value vector for asset 8 = [0.4, 0.3, 0.3]

- PathID for asset 9=[62, 15, 51, 30, 8, 40, 23, 43, 16, 31, 48, 35, 7, 59, 34, 53, 9, 65, 21, 60, 61, 1, 29, 52, 14, 24, 4, 36, 58, 26, 18, 39, 12, 45, 17, 2, 13, 38, 57, 46, 37, 56]
- User service value vector for asset 9 = [0.24, 0.3, 0.18]

- PathID for asset 10=[15, 30, 43, 16, 31, 48, 59, 32, 60, 1, 61, 29, 47, 3, 27, 64, 17, 2, 13, 28, 38, 57, 37, 51, 8, 23, 35, 7, 49, 54, 9, 53, 65, 21, 33, 6, 52, 24, 42, 4, 26, 55, 11, 50, 39, 45, 20, 46, 25, 10, 5]
- User service value vector for asset 10 = [0.4, 0.18, 0.3]

Paths supporting user services:

- The paths supporting user service 0 are = [51, 30, 8, 23, 31, 48, 7, 54, 49, 22, 32, 53, 9, 6, 1, 29, 52, 24, 47, 4, 26, 50, 3, 27, 44, 45, 2, 28, 46, 25, 10, 5, 0]

- The paths supporting user service 1 are = [62, 15, 40, 43, 16, 35, 59, 34, 41, 65, 21, 60, 61, 14, 42, 36, 19, 58, 63, 18, 39, 12, 64, 17, 13, 38, 20, 57, 37, 56]

- The paths supporting user service 2 are = [51, 30, 8, 23, 31, 48, 7, 54, 49, 32, 53, 9, 33, 6, 1, 29, 52, 24, 47, 4, 55, 26, 11, 50, 3, 27, 45, 2, 28, 46, 25, 10, 5]

Annex C

Simulation Scenarios, Results and Learning Agents Parameters

Scenario 1: Simple repair sequence of four (4) outages			
Events	4 outages, all at t=0.		
Concurrent affected assets	4		
Number of NOC staff	1		
Action type choices	2 {fix, wait}		
Learning Task	Find and converge to the optimal sequence of repairs		
State Space	$2^4 = 16$ states		
Solution Space	4! = 24 different paths, infinite with "wait"		
Agent Parameters			
QConnectionist NN policy parameters		**Q-Learning Table policy parameters**	
Gamma (γ)	0.9	**Gamma (γ)**	0.9
Epsilon (ε)	0.1	**Epsilon (ε)**	0.1
Alpha (α)	0.9	**Alpha (α)**	0.9
Lambda (λ)	0.9	**Reward f()**	10 if ∫R(t) ≤ previous objective. Update new optimum. 0 otherwise.
Reward f()	1 if ∫R(t) ≤ previous objective. Update new optimum. 0 otherwise.	**Selection method**	ε-Greedy
Mutate	Yes, 10% of backed-up best-policy after 300 epochs without change. (Genetic Algorithm)	**Epochs**	2000
Selection method	Maximum Q value for output layer. Used 1 hidden layer with 12 neurons.		
Epochs	32000		

Scenario 2: Simple repair sequence of eleven (11) outages			
Events	11 outages, all at t=0.		
Concurrent affected assets	11		
Number of NOC staff	1		
Action type choices	1 {fix}		
Learning Task	Find and converge to the optimal sequence of repairs		
State Space	2^{11} = 2048 states		
Solution Space	11! = 39.9 millions different paths		
Agent Parameters			
QConnectionist NN policy parameters		**Q-Learning Table policy parameters**	
Gamma (γ)	0.9	**Gamma (γ)**	0.9
Epsilon ($\mathcal{E}$)	0.1	**Epsilon ($\mathcal{E}$)**	0.1
Alpha (α)	0.9	**Alpha (α)**	0.9
Lambda (λ)	0.9	**Reward f()**	10 if $\int R(t) \leq$ previous objective. Update new optimum. 0 otherwise. (trialed -10)
Reward f()	1 if $\int R(t) \leq$ previous objective. Update new optimum. 0 otherwise. (trialed -1 as well)	**Selection method**	$\mathcal{E}$-Greedy
Mutate	Yes, 10% of backed-up best-policy after 300 epochs without change. (Genetic Algorithm)	**Epochs**	50 000
Selection method	Maximum Q value for output layer. Used 1 hidden layer with 12 neurons.		
Epochs	100 000		

Results for NN policy (scenario 2)					
Average NN policy	**ζ**	**Average Random policy**	**ζ**	**Average Compared**	**ζ**
13.42075	0.377046	14.87381	0.416481	-1.45305	0.174259
Quality measure:	0.028094	**Quality measure:**	0.028001	**Quality measure:**	-0.11993
CI Max:	13.7978	**CI Max:**	15.29029	**CI Max:**	-1.27879
CI Min:	13.04371	**CI Min:**	14.45732	**CI Min:**	-1.62731

Results for Table policy (scenario 2)					
Average Table policy	**ζ**	**Average Random policy**	**ζ**	**Average Compared**	**ζ**
16.45171	0.435349	17.37847	0.429031	-0.92676	0.11951
Quality measure:	0.026462	**Quality measure:**	0.024688	**Quality measure:**	-0.12895
CI Max:	16.88706	**CI Max:**	17.8075	**CI Max:**	-0.80725
CI Min:	16.01636	**CI Min:**	16.94944	**CI Min:**	-1.04627

Scenario 3: Simple patching sequence of eleven (11) vulnerability instances	
Events	11 vulnerabilities, all at t=0. No exploits.
Concurrent affected assets	11
Number of NOC staff	1
Action type choices	2 {fix, patch} (only patch is valid)
Learning Task	Find and converge to the optimal order of *patching*
State Space	3^{11} = 177 147 states
Solution Space	11! = 39.9 millions different paths

Agent Parameters			
QConnectionist NN policy parameters		**Q-Learning Table policy parameters**	
Gamma (γ)	0.9	**Gamma (γ)**	0.9
Epsilon ($\mathcal{E}$)	0.1	**Epsilon ($\mathcal{E}$)**	0.1
Alpha (α)	0.9	**Alpha (α)**	0.9
Lambda (λ)	0.9	**Reward f()**	10 if $\int R(t) \leq$ previous objective. Update new optimum. 0 otherwise. (trialed -10)
Reward f()	1 if $\int R(t) \leq$ previous objective. Update new optimum. 0 otherwise. (trialed -1 as well)	**Selection method**	$\mathcal{E}$-Greedy
Mutate	Yes, 10% of backed-up best-policy after 300 epochs without change. (Genetic Algorithm)	**Epochs**	50 000
Selection method	Maximum Q value for output layer. Used 1 hidden layer with 12 neurons.		
Epochs	100 000		

Results for NN Policy (scenario 3)					
Average NN Policy	**ζ**	**Average Random**	**ζ**	**Average Compared**	**Z**
1.768807	0.043075	1.869059	0.047671	-0.10025	0.022145
Quality measure:	0.024353	**Quality measure:**	0.025505	**Quality measure:**	-0.22089
CI Max:	1.811882	**CI Max:**	1.91673	**CI Max:**	-0.07811
CI Min:	1.725732	**CI Min:**	1.821388	**CI Min:**	-0.1224
Results for Table Policy (scenario 3)					
Average Table policy	**ζ**	**Average Random**	**ζ**	**Average Compared**	**ζ**
1.60017	0.029754	1.869059	0.047671	-0.26889	0.029206
Quality measure:	0.018594	**Quality measure:**	0.025505	**Quality measure:**	-0.10862
CI Max:	1.629924	**CI Max:**	1.91673	**CI Max:**	-0.23968
CI Min:	1.570416	**CI Min:**	1.821388	**CI Min:**	-0.29809

Scenario 4: Patch assets affected by a new vulnerability or fix exploits.	
Events	1 vulnerability at t=0. One exploit per affected asset at different arrival times.
Concurrent affected assets	≤ 5 (router is the largest "product" group in the model)
Number of NOC staff	1
Action type choices	2, {fix, patch}
Learning Task	Find the optimal order and combination of patching and fixing for different vulnerabilities.
State Space	4^{11} = 4.2 millions states
Solution Space	>>

Agent Parameters

QConnectionist NN policy parameters		Q-Learning Table policy parameters	
Gamma (γ)	0.9	**Gamma (γ)**	0.9
Epsilon (ε)	0.1	**Epsilon (ε)**	0.1
Alpha (α)	0.1	**Alpha (α)**	0.9
Lambda (λ)	0.9	**Reward f()**	10 if ∫R(t) ≤ previous objective. Update new optimum. 0 otherwise.
Reward f()	1 if ∫R(t) ≤ previous objective. Update new optimum. 0 otherwise.	**Selection method**	ε-Greedy
Mutate	Yes, 10% of backed-up best-policy after 300 epochs without change. (Genetic Algorithm)	**Epochs**	50 000
Selection method	Maximum Q value for output layer. Used 1 hidden layer with 12 neurons.		
Epochs	200 000		

Results for Table Policy (scenario 4)					
Average Table Policy	**ζ**	**Average Random**	**ζ**	**Average Compared**	**ζ**
1.613127	0.159042	1.874017	0.185581	-0.26089024	0.123404
Quality measure:	0.098592	**Quality measure:**	0.099029	**Quality measure:**	-0.47301
CI Max:	1.772168	**CI Max:**	2.059598	**CI Max:**	-0.13749
CI Min:	1.454085	**CI Min:**	1.688436	**CI Min:**	-0.38429
Results for NN Policy (scenario 4)					
Average NN policy	**ζ**	**Average Random**	**ζ**	**Average Compared**	**ζ**
1.812247	0.17654	1.874017	0.185581	-0.06176963	0.045728
Quality measure:	0.097415	**Quality measure:**	0.099029	**Quality measure:**	-0.74031
CI Max:	1.988787	**CI Max:**	2.059598	**CI Max:**	-0.01604
CI Min:	1.635707	**CI Min:**	1.688436	**CI Min:**	-0.1075

Scenario 5: Patch, fix, isolate or wait for continuous arrival of vulnerabilities, exploits and outages.	
Events	Continuous arrivals of outages, vulnerabilities and associated spreading exploits.
Concurrent affected assets	≤ 11
Number of NOC staff	1
Action type choices	4 {fix, patch, isolate, wait}
Learning Task	Find the optimal order and combination of patching, fixing, isolating and waiting for different outages, vulnerabilities and exploits states.
State Space	4^{11} = 4.2 millions states
Solution Space	>>

Agent Parameters			
QConnectionist NN policy parameters		**Q-Learning Table policy parameters**	
Gamma (γ)	0.5	**Gamma (γ)**	0.9
Epsilon (ε)	0.1	**Epsilon (ε)**	0.1
Alpha (α)	0.2	**Alpha (α)**	0.8
Lambda (λ)	0.5 Traces reset every time R(t) = 0	**Reward f()**	R(t-1)-R(t) for every actions.
Reward f()	R(t-1)-R(t) for every actions.	**Selection method**	ε-Greedy
Mutate	No.	**Epochs**	1
Selection method	Maximum Q value for output layer. Used 3 hidden layers with 12 neurons.		
Epochs	1		

Results for NN Policy (scenario 5)					
Average Case1	**ζ**	**Average Case2**	**ζ**	**Average Compared**	**Z**
18327.09	297.6208	18893.93	283.7558	-566.842	138.5314
Quality measure:	0.016239	**Quality measure:**	0.015018	**Quality measure:**	-0.24439
CI Max:	18624.71	**CI Max:**	19177.69	**CI Max:**	-428.311
CI Min:	18029.47	**CI Min:**	18610.18	**CI Min:**	-705.374

.Results for Table Policy (scenario 5)					
Average Case1	**ζ**	**Average Case2**	**ζ**	**Average Compared**	**Z**
18402.079	251.49224	18933.875	241.76981	-534.7965	124.13402
Quality measure:	0.013854	**Quality measure:**	0.013079	**Quality measure:**	-0.22555
CI Max:	18651.571	**CI Max:**	19185.645	**CI Max:**	-411.6624
CI Min:	18142.587	**CI Min:**	18694.106	**CI Min:**	-653.9305