



HAL
open science

Detection and identification methodology for multiple faults in complex systems using discrete-events and neural networks: applied to the wind turbines diagnosis

Samuel Toma

► To cite this version:

Samuel Toma. Detection and identification methodology for multiple faults in complex systems using discrete-events and neural networks: applied to the wind turbines diagnosis. Modeling and Simulation. University of Corsica, 2014. English. NNT : . tel-01141844

HAL Id: tel-01141844

<https://hal.science/tel-01141844>

Submitted on 13 Apr 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

UNIVERSITÉ DE CORSE – PASQUALE PAOLI
U.F.R. SCIENCES ET TECHNIQUES

Ph.D Thesis

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Presented by

Samuel TOMA

**Detection and identification methodology for
multiple faults in complex systems using discrete
events and neural networks: *applied to the wind
turbines diagnosis***

Supervisor: **Dominique FEDERICI**

Co-supervisor: **Laurent CAPOCCHI**

Presented on the 8th of September 2014

Committee :

Mr. Humberto HENAO, PRU, *Université de Picardie « Jules Verne »*

Mr. Mamadou Kaba TRAORÉ, PRU, *Université Blaise Pascal Clermont-Ferrand 2*

Mr. Gérard-André CAPOLINO, PRU, *Université de Picardie « Jules Verne »*

Mr. Bernard ZEIGLER, PRU émérite, *Université de l'Arizona*

Mr. Jean-François SANTUCCI, PRU, *Université de Corse « Pasquale Paoli »*

Mr. Dominique FEDERICI, PRU, *Université de Corse « Pasquale Paoli »*

Mr. Laurent CAPOCCHI, MCF, *Université de Corse « Pasquale Paoli »*

UNIVERSITÉ DE CORSE – PASQUALE PAOLI

U.F.R. SCIENCES ET TECHNIQUES

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE CORSE

ÉCOLE DOCTORALE ENVIRONNEMENT ET SOCIÉTÉ

Spécialité : Informatique

présentée par

Samuel TOMA

Méthodologie de détection et d'identification des défauts multiples dans les systèmes complexes à partir d'évènements discrets et de réseaux de neurones : *Applications aux aérogénérateurs*

Directeur de thèse : Dominique Federici

Co-directeur de thèse : Laurent Capocchi

soutenue publiquement le 8 Septembre 2014 devant le jury composé de:

Rapporteurs :

Mr. Humberto HENAO, PRU, *Université de Picardie « Jules Verne »*

Mr. Mamadou Kaba TRAORÉ, PRU, *Université Blaise Pascal Clermont-Ferrand 2*

Examineurs :

Mr. Gérard-André CAPOLINO, PRU, *Université de Picardie « Jules Verne »*

Mr. Bernard ZEIGLER, PRU émérite, *Université de l'Arizona*

Mr. Jean-François SANTUCCI, PRU, *Université de Corse « Pasquale Paoli »*

Mr. Dominique FEDERICI, PRU, *Université de Corse « Pasquale Paoli »*

Mr. Laurent CAPOCCHI, MCF, *Université de Corse « Pasquale Paoli »*

Abstract

This thesis deals with the time-domain analysis of the electrical machines fault diagnosis due to early short-circuits detection in both stator and rotor windings. It also introduces to the Discrete Event system Specification (DEVS) a generic solution to enable concurrent and comparative simulations (CCS). The DEVS-based CCS is an extension introduced using an aspect-oriented programming (AOP) to interact with the classic DEVS simulator. A new DEVS-based artificial neural network (ANN) is also introduced with a separation between learning and calculation models. The DEVS-based CCS is validated on the proposed ANN DEVS library inside the DEVSimPy environment. The concurrent ANN contributes in the time-domains analysis for the electrical machine fault diagnosis. This new method is based on data coming directly from the sensors without any computation but with a new dedicated pre-processing technique. Later, some enhancements are brought to the artificial neural network based on a new multistage architecture reducing the training time and errors compared to the single ANN. The new architecture and techniques has been validated on real data sixteen non-destructive windings faults analysis and localization.

Achnowledgments

This PhD has been a truly life-changing experience for me and it would not have been possible without the support and guidance that I received from many people.

First and foremost, I wish to thank my advisor, professor Dominique Federici for his patient guidance, encouragement, advice and assistance he provided at all levels of this research project. He helped me to focus on the project goal and its credibility. Beside my advisor I would like to thank my co-advisor Dr. Laurent Capocchi, he has always been supportive since my undergraduate studies in 2010. I would like to thank him for encouraging my research and for allowing me to grow as a research scientist. His guidance helped me during the hole research project and during the writing of this thesis. He has always encouraged me when I failed, and kept telling me that this is the only way to success. His advice on both research as well as on my career have been priceless, without his help, patience, constant feedback and availability, this PhD would not have been achievable.

I am also grateful to professor Gérard-André Capolino, his insightful comments and constructive criticisms, at different stages of my research, were thought-provoking and they helped me focus my ideas. I am grateful to him for holding me to a high research standard and enforcing strict validations for each research result, and thus teaching me how to do research. I appreciate his vast knowledge and skill in many areas. I would never forget professor Richard Grisel, who helped me through the FPGA field which is important to the perspectives of this thesis.

I would like to thank all the SPE laboratory team members, directed by professor Paul-Antoine Bisgambiglia for the great working environment. A special thanks to professor Jean-François Santucci the leader of the TIC project for the healthy research environment he created among us and for encouragement he gave me during my working days. I would like to thank the various members of this project, they provided a friendly and cooperative working atmosphere as well as useful feedback and insightful comments on my work.

My sincere thanks also goes to Dr. Marie-Laure Nivet and Dr. Christophe Paoli for their help and support during my teaching activities beside my research work. Never to forget Dr. Fabien Flori how gave the opportunity to start my first collaboration with the University of Corsica during my master degrees. He has always been their

when i needed help.

I gratefully acknowledge the funding received towards my PhD from the Doctoral school of the university of Corsica, "Environnement et société" directed by the professor Jean Costa.

Finally, I would like to acknowledge my friends and family who supported me during my working days. I consider my self lucky to have such family and friends.

Résumé

L'étude présentée dans ce mémoire concerne le diagnostic des machines électriques à l'aide d'une association innovante entre la modélisation à événements discrets, la Simulation Comparative et Concurrente (SCC) et les Réseaux de Neurones Artificiels (RNAs). Le diagnostic des machines électriques est effectué à partir d'une analyse temporelle des signaux statoriques et rotoriques à l'aide de réseaux de neurones de type Feed-Forward. Afin de faciliter la configuration de ces réseaux de neurones, l'approche proposée dans ce document utilise la simulation comparative et concurrente implémentée grâce au formalisme à événements discrets DEVS (Discrete Event system Specification). L'intégration des algorithmes de la SCC et des RNAs au sein du formalisme DEVS a été effectuée de manière générique en développant des plug-ins et une bibliothèque de modèles dans l'environnement de modélisation et de simulation à événements discrets DEVSIMPy. L'application de cette nouvelle solution pour le diagnostic des machines électriques permet de détecter les défauts à partir d'une architecture logiciel facilement portable sur des systèmes embarqués de type FPGA.

Introduction

Les techniques de surveillance et de diagnostics des machines électriques sont en constante évolution depuis les années 80. Elles ont commencé par une simple analyse humaine pour arriver jusqu'aux processus de décision modernes impliquant des techniques de traitement de signaux. Ce n'est que très récemment que ces techniques sont intégrées dans les systèmes électromécaniques dans lesquels les machines électriques sont utilisées comme des moteurs ou des générateurs. Depuis ces vingt dernières années, les machines asynchrones à courant triphasé à cage d'écureuil, particulièrement intéressantes pour leur fiabilité et leur prix, ont été le sujet de plusieurs études et travaux de recherche concernant leur diagnostic. Il existe un nombre important de revues scientifiques établissant un état de l'art des techniques de diagnostics des machines électriques avec une attention particulière pour les défauts électriques [1–4]. Ces travaux expliquent de manière exhaustive comment, à partir d'une modélisation de la machine électrique, des processus de décision permettent d'établir un diagnostic à partir d'instruments de mesure et de traitements des signaux. Dans le domaine des techniques de surveillance et de diagnostic des machines électriques, les aérogénéra-

teurs sont un élément-clé fondamental pour la détection des défauts. Actuellement, beaucoup de projets sont aux prises des techniques de surveillances des parcs éoliens et quelques-uns sont même à l'origine de standard. Cependant, ces technologies nécessitent encore d'évoluer afin d'obtenir des solutions techniques génériques et efficaces qui répondent aux objectifs indispensables de fiabilité [5]. En effet, la plupart des parcs éoliens modernes sont fondés sur la technologie des générateur à induction à rotor bobiné (WRIG) avec un nombre faible de pôles et un réducteur planétaire pour adapter la vitesse de l'arbre du rotor avec la vitesse des lames de l'éolienne. Même si il y a un grand nombre de travaux scientifiques qui s'intéressent au contrôle des machines à induction, le nombre de travaux qui s'intéressent à la détection et à la localisation des défauts reste faible [3, 4, 6]. Parmi ces travaux, les techniques de traitement de signaux sont serte capables d'identifier des défauts électriques de nature différente mais elles sont souvent qualifiées de complexes. En effet, il est très difficile de donner une conclusion précise sur des défauts mis en évidence à partir d'une analyse temporelle, fréquentielles ou temps-fréquence du fait de la complexité des signaux (périodicité, bruit) issues du phénomène observé. Les chercheurs se sont alors tournés vers des techniques à base de réseaux de neurones qui sont réputées pour donner des résultats remarquables au détriment d'une configuration du réseaux de neurones souvent obtenue par des approches non déterministes [7–9]. Les réseaux de neurones ont fait leur apparition au sein des techniques de diagnostic il y a maintenant à peu près une vingtaine d'années [10]. Ils ont été appliqués avec succès sur les machines à induction triphasé pour détecter des fautes du coté du rotor [11–13]. Dans le pratique, les réseaux de neurones ont toujours été utilisés pour améliorer la classification des défauts et pour éviter des fausses alertes ou des dysfonctionnements évidents.

Le formalisme DEVS (Discrete Event system Specification) a été introduit par le professeur B.P. Zeigler en 1976 [14]. Ce formalisme permet une modélisation modulaire et hiérarchique des systèmes complexes à évènements discrets. DEVS propose deux types de modèles pour modéliser un système : les modèles atomiques et les modèles couplés. Un modèle atomique DEVS est vu comme une boîte noire qui reçoit et émet des évènements (messages). Il permet de spécifier le comportement d'un système. Le modèle couplé permet de représenter la structure d'un système. Il peut être composé de modèles atomiques et/ou couplés. Le simulateur d'un modèle DEVS est

généralisé automatiquement par le formalisme sur la base d'un arbre de simulation abstrait construit à partir des modèles atomiques et des modèles couplés qui compose le modèle. Un des points forts du formalisme DEVS réside dans sa capacité à accepter des évolutions à la fois du côté de la modélisation et de la simulation. En effet, DEVS repose sur une séparation explicite entre la modélisation et la simulation d'un système. Toutes les extensions de modélisation/simulation qui respectent le formalisme assurent la compatibilité avec le simulateur/modèle. Le projet libre sous licence GPL v3 DEV-SimPy [15] est un environnement de modélisation et de simulation DEVS implémenté en langage Python. Tous les travaux présentés dans ce mémoire sont implémentés dans DEV-SimPy. C'est un environnement modulaire qui propose un système de plug-ins qui, couplé avec une programmation orientée objets (POO) et orientée aspects (POA), permet d'étendre ses fonctionnalités. Il propose également une gestion des modèles sous forme de bibliothèques dynamiques. Le simulateur DEVS est séquentiel et il exécute donc les modèles, suivant un ordre précis, les uns après les autres. Cependant, il est possible d'exécuter des simulations DEVS de manière concurrente et comparative à condition d'intégrer les algorithmes de la Simulation Comparative et Concurrente (SCC) au sein du formalisme DEVS. La puissance essentielle de la SCC réside dans le fait que plusieurs expériences sont simulées d'une manière implicite au travers d'une seule simulation. L'intégration de la SCC dans le formalisme DEVS a déjà été présentée dans le cas de la simulation des défauts comportementaux dans les systèmes digitaux et a été proposée sous le nom de BFS-DEVS (Behavioral Fault Simulator for DEVS) [16]. Du fait de la séparation explicite entre la modélisation et la simulation dans le formalisme DEVS et du système de plug-ins proposé dans DEV-SimPy, l'intégration de la SCC avec DEVS est rendue possible par l'implémentation d'un plug-in sans modifier le noyau de simulation DEVS. Grâce à ce plug-in, l'utilisateur peut configurer et exécuter plusieurs simulations concurrentes dans l'environnement DEV-SimPy à partir du moment où il définit un comportement concurrent au sein de ces modèles atomiques.

Les réseaux de neurones artificiels (RNAs) sont très utilisés pour modéliser des systèmes indescriptibles de manière déterministe par les mathématiques. Les réseaux de neurones déduisent par apprentissage le comportement d'un système en cherchant une correspondance entre les entrées et les sorties du système. Cette représentation

de type boîte noir est proche de DEVS et il serait intéressant de posséder la technique des RNAs dans DEVS afin de modéliser des phénomènes dont on ne connaît pas la description mathématique mais dont on possède les événements d'entrée et de sortie. L'approche de modélisation des RNAs dans DEVS pose un certain nombre de questions comme le niveau de description choisi dans DEVS pour modéliser un neurone ou le niveau de modularité entre l'apprentissage et le test du réseau de neurones. Une librairie nommée NN a été implémentée dans l'environnement DEVSimPy afin de proposer la modélisation des RNAs dans le formalisme DEVS. La principale difficulté émanant de l'utilisation des RNAs réside dans leur configuration. Cette dernière repose principalement sur un processus itératif consistant à essayer plusieurs paramètres de configuration (momentum, fonction d'activation, nombre de neurones cachées, etc.) afin d'obtenir une convergence à la fois dans l'apprentissage et dans le test des RNAs. Les travaux présentés dans ce manuscrit montrent qu'il est possible d'avoir recours à la SCC pour accélérer la configuration des RNAs en simulant plusieurs configurations concurrentes en une seule simulation DEVS.

En conclusion, l'idée principale des travaux présentés regroupe les trois concepts DEVS, RNA et SCC autour d'une problématique définie par le diagnostic des machines électriques à partir d'une modélisation à base de RNAs configurés en utilisant la SCC le tout dans un cadre formel assuré par le formalisme DEVS.

Ce document est organisé de la manière suivante : après une introduction qui positionne le problème général de notre étude, le premier chapitre présente un état de l'art des précédents domaines étudiés (DEVS, RNA, SCC, WRIG). Le deuxième chapitre présente la librairie NN DEVS permettant de modéliser un réseau de neurones artificiels de type Feed-Forward. Le troisième chapitre présente l'intégration de la simulation comparative et concurrente dans le formalisme DEVS à partir d'une extension de la modélisation en préservant le noyau de simulation. Le quatrième chapitre présente l'implémentation et la configuration des RNAs DEVS à partir des algorithmes de la SCC. Le cinquième et dernier chapitre propose d'appliquer les précédentes techniques et concepts à la problématique de détection des défauts dans les machines électriques de type WRIG. Enfin, une conclusion résume le travail réalisé dans cette thèse ainsi que les perspectives des travaux.

Les réseaux de neurones modélisés avec DEVS

Les réseaux de neurones sont souvent représentés par des boîtes noires dont le comportement est construit à partir de la connaissance des entrées et des sorties. Une fois l'apprentissage effectué, ces systèmes peuvent être utilisés pour faire de la prédiction ou bien de la classification de données. Dans cette thèse, nous étudierons les RNAs pour classifier des défauts pouvant apparaître au sein des machines électriques. L'architecture d'un RNA est constituée d'une couche d'entrée, d'une ou de plusieurs couches cachées et d'une couche de sortie. Chaque couche possède un certain nombre de neurones (cf. Figure 1) qui lorsqu'ils sont associés permettent d'apprendre le comportement du système modélisé. La couche d'entrée des réseaux de neurones reçoit une liste de messages qui contiennent toutes les entrées du réseau. Ensuite ces messages d'entrées sont transmis à la couche cachée. La couche cachée calcule pour chacun des neurones, une fonction de combinaison suivie d'une fonction d'activation (cf. Figure 1) et envoie à son tour le résultat à la couche suivante.

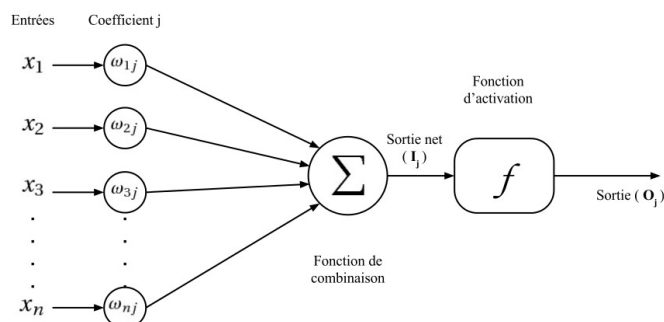


FIGURE 1 – L'architecture d'un neurone artificiel j .

Parmi les types de RNA, le perceptron multicouche est un RNA organisé de plusieurs couches au sein desquelles les informations circulent de la couche d'entrée vers la couche de sortie uniquement dans un mouvement unidirectionnel. Les premiers réseaux de neurones n'étaient pas capables de résoudre des problèmes non linéaires. C'est seulement avec l'apparition de la rétro-propagation du gradient de l'erreur que cette limitation a été supprimée [17].

Les RNAs ont une architecture intrinsèquement liée à celle des modèles à événements discrets. Il existe de manière évidente une similitude entre les couches, les neurones et la notion de boîte noire transformant des données d'entrées en données de sorties. Nous avons proposés une librairie de modèles DEVS dans le logiciel DEV-

SimPy afin de modéliser les RNAs. Cette librairie, nommé NN pour *Neural Network*, a fait l'objet d'une étude au préalable afin de déterminer une architecture modulaire et performante. Cette étude a pris en compte les facteurs suivants : l'accès à la liste des données d'entrées et de sorties, le choix de l'architecture d'un neurone artificiel, les différents liens entre les neurones des couches et la possibilité de modifier l'algorithme d'apprentissage.

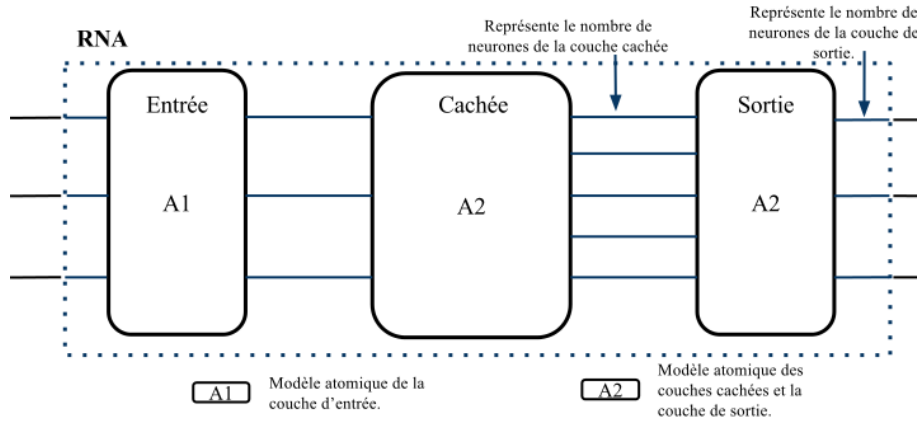


FIGURE 2 – Modélisation DEVS d'un RNA.

L'étude de ces quatre points ont conduit à une combinaison des modèles atomiques qui sont présentés sur la figure 2. Dans cette approche, un modèle atomique DEVS permet de modéliser une couche d'un RNA. Le modèle atomique A1 permet de représenter la couche d'entrée, le modèle atomique A2 la couche cachée et la couche de sortie. Plusieurs avantages sont à souligner avec cette approche de modélisation : le nombre de neurones dans chaque couche est visible à tout moment de la simulation par le nombre de sortie de chaque couche, le regroupement de tout les neurones d'une couche dans un seul modèle atomique facilite la configuration de ces neurones en configurant qu'un seul modèle, les connections entre les couches peuvent être facilement modifiées (suppression ou redirection).

Ce choix d'architecture nous conduit à une conception complète d'un réseau de neurones avec sa phase d'apprentissage comme présenté sur la figure 3. L'algorithme d'apprentissage présenté est connu sous le nom de rétro-propagation [18–20].

La figure 3 présente la modélisation d'un RNA basée sur l'utilisation de modèles atomiques DEVS. Dans cette figure ce trouve quatre bloque qui groupent quatre catégories différentes de modèles atomiques. Le premier bloque se compose d'un seul modèle atomique (*Input*) qui modélise le comportement de la couche d'entrée du ré-

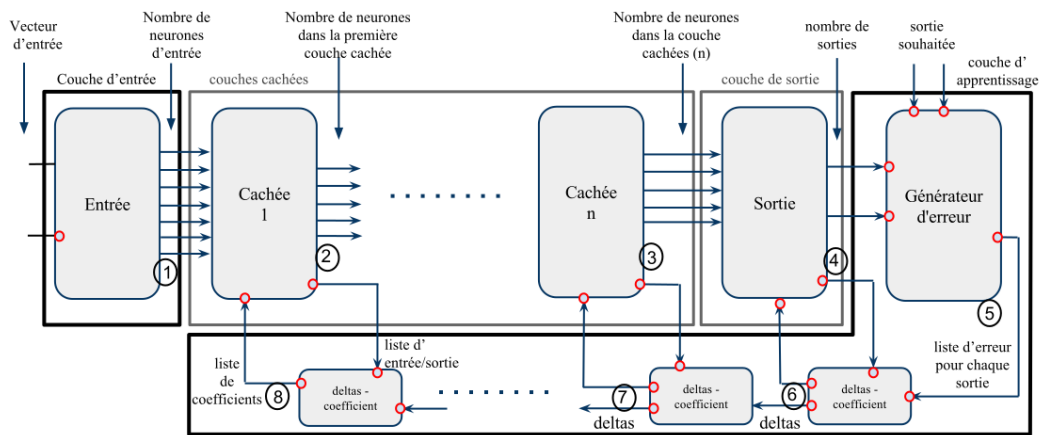


FIGURE 3 – Modélisation DEVS d'un RNA avec la phases d'apprentissage.

seau. C'est la couche d'entrée qui contrôle le flux des données qui circulent dans le réseau (cf. étape 1 sur la figure 3). Le deuxième bloque présente un modèle atomique (*Hidden*) par couche cachée (cf. étapes 2 et 3 sur la figure 3). Si le RNA possède plusieurs couche cachées, celles-ci seront représentées par une interconnexion de modèles atomiques du type *Hidden*. Le quatrième bloque représente la couche de sortie avec le modèle atomique *Output* (cf. étape 4 sur le figure 3) qui génère la sortie du RNA. Dans notre approche, la sortie de cette couche est envoyée à un modèle atomique appelé générateur d'erreurs qui représente la phase d'apprentissage du réseau. La sortie du générateur d'erreurs (cf. étape 5 sur la figure 3) est calculée en faisant la différence entre la sortie réelle du réseau et la sortie réelle du système. Les dernières étapes (cf. étapes 6, 7 et 8 sur la figure 3) représentent la rétro-propagation de l'erreur et la modification des coefficients de calcul de chacune des couches de calcul (couche cachée ou de sortie).

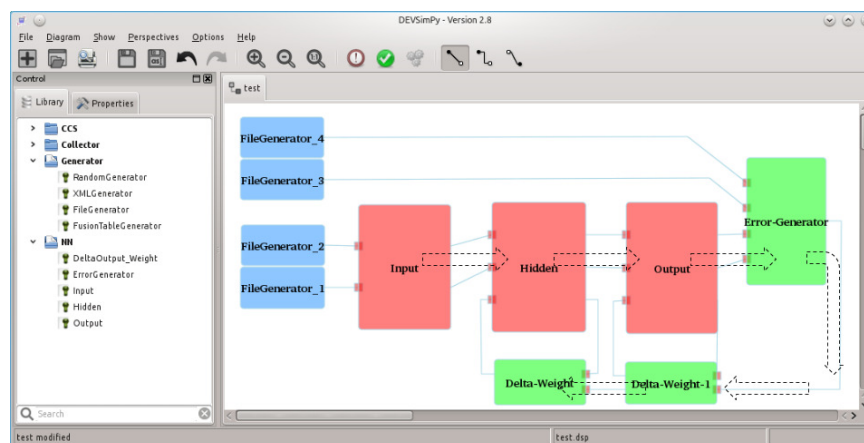


FIGURE 4 – Un réseau de neurones implémenté dans DEVSIMPy.

La figure 4 présente la modélisation d'un RNA dans le logiciel DEVSimPy. La librairie NN sur le panel de gauche contient l'ensemble des modèles nécessaires pour la modélisation d'un RNA du type Feed-Forward. Nous retrouvons les modèles présentés auparavant comme *Hidden* pour la couche cachée, *Output* pour la couche de sortie et *Input* pour la couche d'entrée. Pour instancier l'un de ces modèles, l'utilisateur doit procéder à un glisser/déposer à partir de la librairie NN vers le panel de droit. Le panel de droit présente la modélisation d'un RNA avec : des générateurs d'évènements (*FileGenerator*), une couche d'entrée (*Input*), une couche cachée (*Hidden*), une couche de sortie (*Output*) et un ensemble de modèles pour l'apprentissage (*Error_Generator*, *Delta_Weight*). Après la simulation, si l'erreur d'apprentissage diverge, l'utilisateur peut modifier les paramètres des modèles atomiques et relancer la simulation jusqu'à obtenir la convergence du RNA.

La simulation comparative et concurrente avec le formalisme DEVS

L'idée d'avoir une Simulation Comparative et Concurrente (SCC) avec le simulateur DEVS a déjà été introduite avec l'extension BFS-DEVS (Behavioral Fault Simulation for DEVS). Cette extension a été réalisée pour la simulation des fautes dans les systèmes digitaux grâce à une modification du noyau de simulation DEVS. Cette modification a permis de prendre en compte des comportements fautifs définis dans les modèles atomiques par l'intermédiaire d'une nouvelle fonction de transition (δ_{fault}). BFS-DEVS utilise des messages fautifs pour communiquer entre les modèles atomiques et traiter leur changement d'état fautif. Dans nos travaux nous proposons de généraliser BFS-DEVS en intégrant les algorithmes de la SCC dans le formalisme avec une implémentation orientée aspect. Contrairement à BFS-DEVS, le noyau de simulation DEVS reste inchangé et le comportement concurrent d'un modèle atomique est géré uniquement coté modélisation par l'introduction d'une fonction concurrente (f_{conc}) défini par l'utilisateur. Une extension utilisant le concept de la programmation orientée par aspect va ensuite générer le comportement d'un modèle atomique compatible avec le noyau de simulation DEVS d'origine. Nous proposons une autre extension pour manipuler les SCC pendant la simulation DEVS. Grâce à l'utilisa-

tion des patrons de conception UML (Unified Modeling Language) (comme le patron Singleton), les SCC sont gérées de manière générique et l'utilisateur peut insérer, modifier ou supprimer des SCC pendant la simulation. Toutes ces propositions ont été implémentées en langage Python dans l'environnement DEVSimPy.

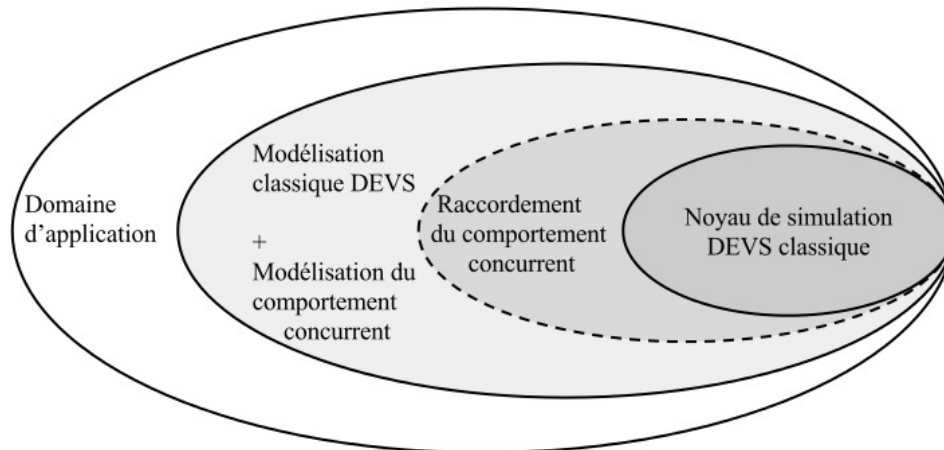


FIGURE 5 – Coresspondance de la simulation comparative et concurrente avec DEVS.

Définition

La figure 5 représente les différentes couches entre le simulateur DEVS et un domaine d'application pour une simulation comparative et concurrente. Pour la simulation concurrente d'un système donné, il faut réaliser la modélisation classique des modèles DEVS avec la modélisation des comportements concurrents. La modélisation d'un modèle atomique avec un comportement concurrent est réalisée grâce à la structure décrite ci-dessous :

$$AM' = \langle X, Y, S, \{H_n\}, \delta_{int}, \delta_{ext}, f_{conc}, \lambda, t_a \rangle$$

avec :

- $X = \{(p, v) \mid p \in in_ports, v \in X_p\}$ la liste des ports et des valeurs d'entrée ;
- $Y = \{(p, v) \mid p \in out_ports, v \in Y_p\}$ la liste des ports et des valeurs de sortie ;
- S l'ensemble des variables d'états ;
- H_n l'ensemble des sous-signatures où :

- $H_n = \{(X_n, Y_n, v_n)\}$;
- Where:
 - * X_n est l'ensemble des ports et de valeurs d'entrée pour une expérience n ,
 - * Y_n est l'ensemble des ports et de valeurs de sortie pour une expérience n ,
 - * v_n est l'ensemble des variables privées pour une expérience n ,
 - * $n \in \mathbb{N}$ ou $0 \leq n \leq N$ avec N le nombre d'expériences par simulation.
- $\delta_{int} : S \rightarrow S$ la fonction de transition interne ;
- $\delta_{ext} : Q \times X \rightarrow S$ la fonction de transition externe avec :
 - $Q = \{(s, e) \mid s \in S, 0 \leq e \leq t_a(s)\}$ l'ensemble des états ;
 - e le temps écoulé depuis la dernière transition.
- $f_{conc} : H_n \rightarrow H_n$ la fonction du comportement concurrent ;
- $\lambda : S \rightarrow Y$ la fonction de sortie ;
- $t_a : S \rightarrow \mathbb{R}_{0,+}^+$ le temps de vie de l'état S , $t_a \in [0, \infty[$.

La nouvelle structure du modèle atomique, aura une simulation basée sur le déclenchement des fonctions comme montré sur la figure 6. Selon le système modélisé, la fonction du comportement concurrent sera liée à une des fonctions de transition interne ou externe. Pour un modèle de type générateur la fonction principale est de générer des données (fonction de transition interne) puis de les envoyer pendant l'exécution de la fonction de sortie. Dans ce cas, la fonction du comportement concurrent sera déhanchée après la fonction de transition interne.

Dans un autre cas, le récepteur de donnée (par exemple un collecteur d'évènements) ne réagit que pour recevoir les données, ce qui implique que la liaison concurrente est appliquée entre la fonction de transition externe et la fonction du comportement concurrent (cf. figure 6).

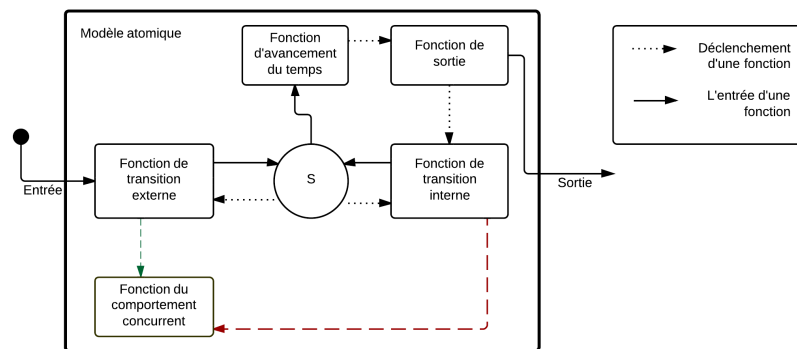


FIGURE 6 – La simulation du comportement concurrent d'un modèle atomique DEVS.

Implémentation

Pour réaliser ces simulations comparatives et concurrentes avec le noyau de simulation classique DEVS nous avons réalisé les étapes suivantes :

1. Création d'une signature pour chaque modèle atomique qui se compose de plusieurs sous-signatures présentant les données des expériences simulées ;
2. Implémentation de la fonction du comportement concurrent ;
3. Réalisation du lien entre la fonction de comportement concurrent et la fonction de transition choisie ;
4. Gestion de la simulation concurrente (ajout, suppression ou modification des expériences).

Pour réaliser ces étapes, plusieurs classes ont été implémentées dans le langage de programmation Python. Python est un langage dynamique et orienté objet qui favorise également la programmation orientée aspect. Un diagramme UML des classes implémentées est présentée sur la figure 7.

Trois catégories de classes sont considérées dans le diagramme UML : le groupe des modèles atomiques, le groupe des signatures, le groupe des gestionnaires. Le groupe des modèles atomiques se compose d'un modèle atomique classique, d'interface et d'un décorateur du modèle atomique. C'est avec le décorateur que le lien entre la fonction du comportement concurrent et les fonctions de transition est réalisé. Le groupe des

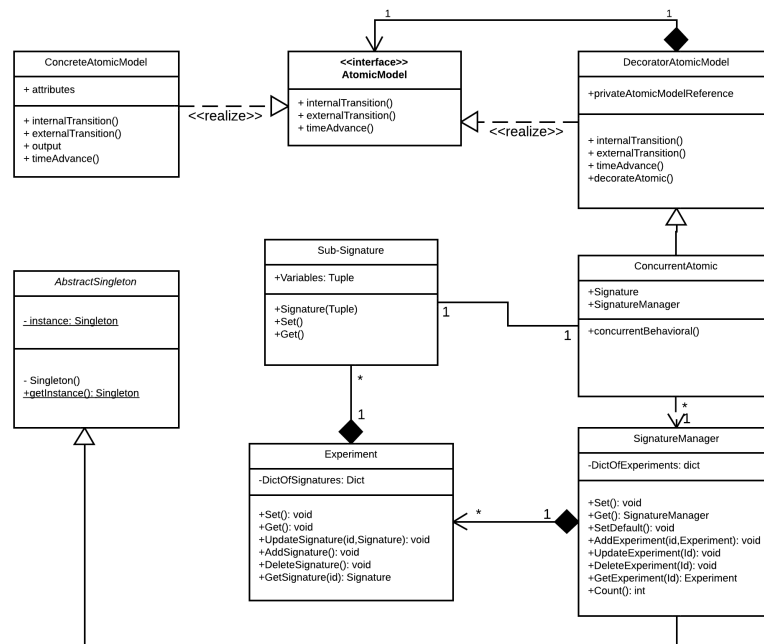
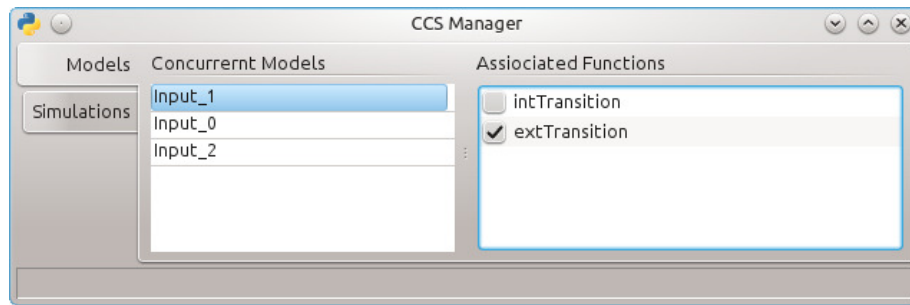


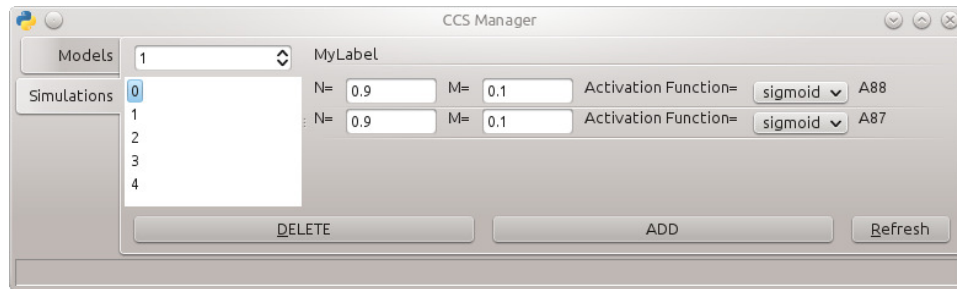
FIGURE 7 – Diagramme de classes UML pour l’implémentation de la SCC dans DEV-SimPy.

signatures introduit la base de données à l’intérieur de la quelle les signatures et les sous-signatures des modèles concurrents sont stockées. Le troisième groupe de classes se compose seulement de la classe *SignatureManager* qui est une classe responsable de la gestion des expériences en concurrence : ajouter, supprimer, modifier ou comparer des expériences pendant une simulation. La figure 8 représente l’interface graphique du plug-in global réalisé dans le logiciel DEVSimPy pour configurer la simulation comparative et concurrente du réseaux de neurones ainsi que pour la configuration des modèles concurrents.

Toutes ces classes sont implémentées dans DEVSimPy en utilisant la notion d’extension globale disponible dans l’environnement. L’affectation de la fonction concurrente se fait par décoration des fonctions de transition. La classe du *DecoratorAtomicModel* (cf. figure 7) sera implémentée dans une extension globale du logiciel DEVSimPy (cf. figure 8). L’utilisateur qui veut utiliser la SCC n’aura qu’à activer l’extension dans DEVSimPy. Sur la figure 8(a), l’interface de dialogue permet de sélectionner les fonctions de transition DEVS (la fonction de transition externe sur l’exemple de la figure 8a) associées à la fonction concurrente des modèles atomiques



(a) Configuration des modèles concurrents.



(b) Configuration des simulations concurrentes.

FIGURE 8 – Interfaces de l’extension global pour la configuration du comportement concurrent.

(*Input_1* sur l’exemple de la figure 8(a)). La figure 8(b) présente l’interface de dialogue permettant d’interagir avec le gestionnaire de simulations concurrentes. Dans l’exemple, la simulation 0 est en concurrence avec les simulations 1, 2, 3 et 4 (partie gauche de la figure 8(b)). Pour cette simulation 0, les modèles A88 et A87 sont configurés (N, M Activation Function) comme il est montrée sur la partie droite de la figure 8(b).

La figure 9 représente un diagramme de séquence qui explique les huit étapes qui composent le déroulement des actions pendant une simulation concurrente avec DEVSimPy. La figure 9 présente trois numérotations sur la ligne du *Gestion de modélisation* qui représente l’interaction de l’utilisateur avec DEVSimPy. L’utilisateur lance la simulation (la simulation concurrente est transparente pour lui), il peut mettre la simulation en pause pour changer quelques configurations de simulation et la reprendre sans s’occuper de la gestion des simulations concurrentes.

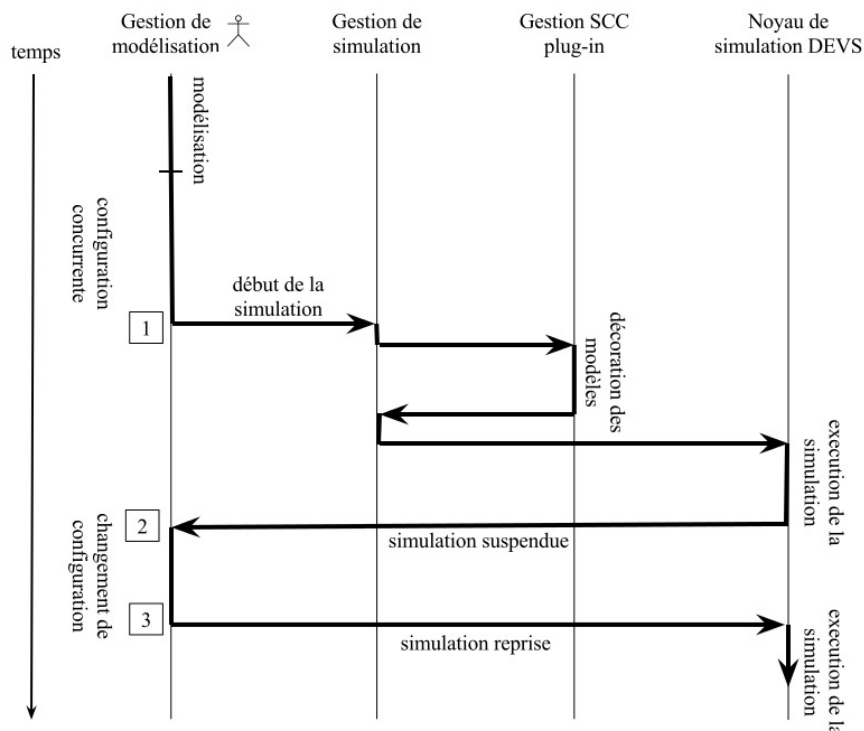


FIGURE 9 – Diagramme de séquences de la simulation comparative et concurrente dans DEVS.

Les réseaux de neurones avec le formalisme DEVS et la SCC

Comme il est montré sur la figure 10, les deux approches introduites dans les sections précédentes (les réseaux de neurones avec le formalisme DEVS, la simulation comparative et concurrente avec le formalisme DEVS) ont été utilisées pour proposer une solution générique basée sur le formalisme DEVS permettant de modéliser des systèmes à l'aide de réseaux de neurones configurables par la SCC. Cette solution sera mise en œuvre dans le cadre d'une application concrète consistant à modéliser et à simuler des machines électriques pour établir leur diagnostic.

L'idée est de compléter le comportement des modèles atomiques de la librairie NN en ajoutant une fonction concurrente permettant de modéliser différentes configurations (paramètres) d'une couche d'un RNA. Ensuite, la SCC permettra pendant la simulation de tester plusieurs configurations possibles du réseau de neurones simulé. L'algorithme 0.1 présente le code Python de la fonction concurrente correspondant au modèle *Input* de la librairie NN.

La ligne 2 de l'algorithme 0.1 vérifie si le modèle est dans l'état actif. Dans ce

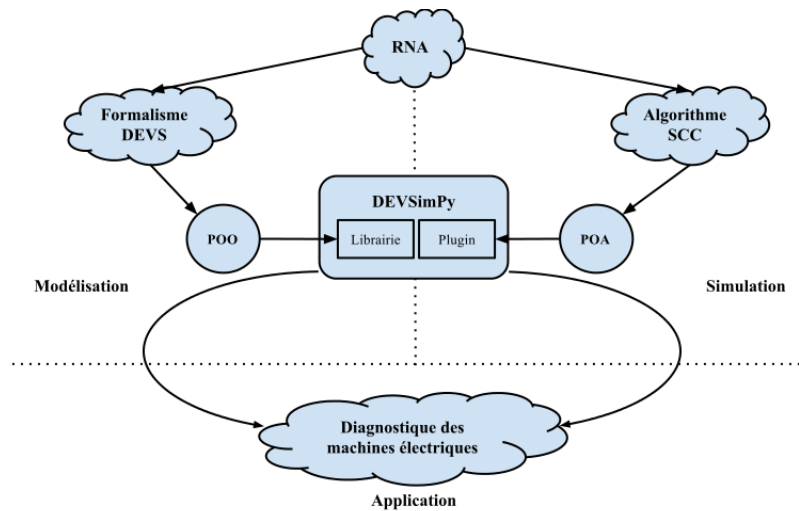


FIGURE 10 – Stratégie du travail réalisé.

Algorithme 0.1 La fonction du comportement concurrent de la couche d'entrée d'un RNA.

```

1 def concBehavFunc(self):
2     if self.state['status'] == 'ACTIVE':
3         Singleton = self.simData.getInstance()
4         for exp in self.expDico:
5             subSignature = Singleton[exp][self.myID]:
6             subSignature['outputs'] = subSignature['inputs']
7                                     + subSignature['Sum']

```

cas, le modèle fait appel au gestionnaire de la simulation concurrente *Singleton* puis ajoute sa signature dans la base de données.

L'utilisation de la SCC pour la configuration d'un RNA dans le formalisme DEVS n'apporte aucune nouveauté au niveau conceptuel. Cependant, d'un point de vue de l'implémentation, deux points sont à souligner. La solution proposée a été possible grâce à la hiérarchie et la modularité du formalisme DEVS et à la possibilité d'extension du logiciel DEVSimPy.

Détection des défauts dans les génératrices à induction à rotor bobiné

Cette section montre une application concrète mettant en œuvre le regroupement des différents domaines abordés dans cette thèse. Le but est de modéliser et de simuler le diagnostic d'une génératrice à induction à rotor bobiné à l'aide des RNA et du

formalisme DEVS. Pour cela nous diviserons le travail en trois parties : la préparation des données à l'aide d'un algorithme de compression travaillant à partir des données numériques, l'utilisation des réseaux de neurones concurrents pour trouver la configuration qui conduit à la convergence de l'erreur d'apprentissage et de test, la proposition d'une méthode générique d'optimisation de classification de données basée sur des calculs statistiques et sur une architecture multi-étages de réseaux de neurones.

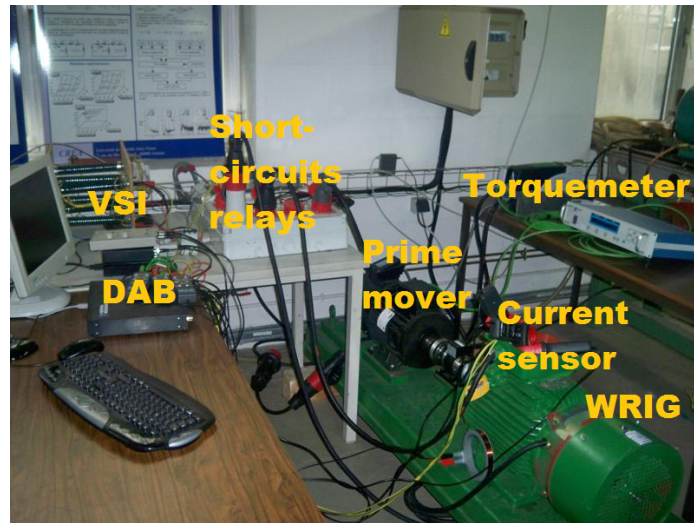


FIGURE 11 – Banc de test.

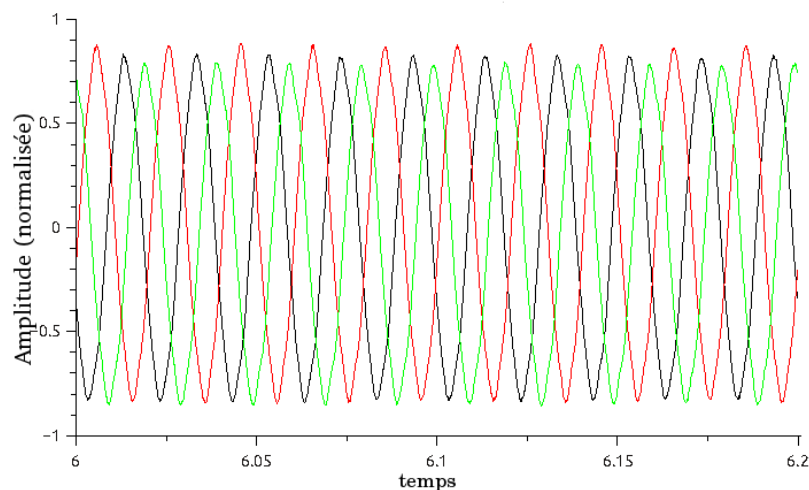


FIGURE 12 – Échantillons des trois courants statoriques en régime permanent.

L'installation présentée sur la figure 11, a été mise en place pour réaliser des mesures sur une génératrice à induction à rotor bobiné triphasée de 5,5kW, 50Hz,

220/380V, 8-pôles. La génératrice est alimentée par une motrice autour d'une machine à induction à cage d'écurueil triphasé de 7kW et d'une tension programmable (onduleur statique de 11kW). Pour simuler la vitesse du vent et faire tourner le système sous différents mode de fonctionnement, la motrice est contrôlée. Deux états de fonctionnement sont disponibles : un état stationnaire et un état transitoire qui simule les différents états du vent. Le codage des données se fait avec une précision de 16 bits. L'installation est réalisée dans le but de pouvoir collecter les courants statoriques et rotoriques de la génératrice. La figure 12, représente un échantillon des courants statoriques normalisés. Les défauts életriques étudiées sont nommées de F1 à F16 :

Côté rotor :

- F1 : court-circuit sur les deux premières spires des deux premières phases, et sur la première spire de la troisième phase.
- F2 : court-circuit sur les deux premières spires des trois phases.
- F3 : court-circuit sur les deux premières spires de la premier phase, et la première spire de la deuxième phase.
- F4 : court-circuit sur les deux premières spires des deux premières phases.
- F5 : court-circuit sur la deuxième spire de la première phase.
- F6 : court-circuit sur les deux premières spires de la première phase.

Côté stator :

- F7 : court-circuit sur la premier spire des deux premières phases.
- F8 : court-circuit sur la deuxième spire des deux premières phases.
- F9 : court-circuit sur la deuxième spire de la première phase.
- F10 : court-circuit sur la deuxième spire de la deuxième phase.
- F11 : court-circuit sur les deux premières spires de la deuxième phase.
- F12 : court-circuit sur les deux premières spires de la deuxième phase.
- F13 : court-circuit sur la première spire de la deuxième phase.
- F14 : court-circuit sur la première spire de la première phase, et le deuxième spire de la deuxième phase.
- F15 : court-circuit sur la deuxième spire de la première phase, et la première spire de la deuxième phase.

- F16 : court-circuit sur la première spire des deux premières phases.

La préparation des données est une phase indispensable pour une utilisation des réseaux de neurones. Deux facteurs peuvent influencer le choix de la technique employée pour la préparation des données : le type de données, la dimension des données. Le type de données collectées du moteur électrique qui nous intéresse est périodique. Les courants rotoriques et statoriques sont composés de plusieurs fréquences. La différence des fréquences est un critère très important lorsqu'on parle de compression des données. Les données d'entrée sont représentées par des vecteurs qui alimente ensuite les réseaux de neurones. Dans notre cas, un vecteur d'entrée est composé des trois courants rotoriques et des trois courants statoriques observés pendant 0,2s avec une période d'échantillonnage de 0.1ms. Ce qui implique un vecteur d'entrée de 12,000 valeurs. Avec une telle dimension, une compression est indispensable sous peine de ne jamais atteindre le critère de convergence du RNA.

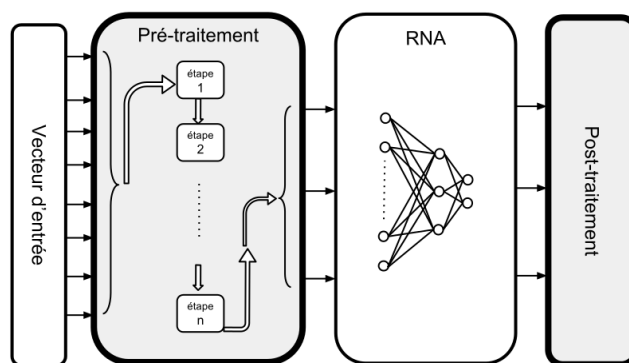


FIGURE 13 – Préparation des données avant et après le réseau de neurones.

La figure 13, présente l'utilisation d'un RNA avec les deux étapes de préparation des données en entrée et en sortie. La première étape est la normalisation des données pour éviter une saturation de la fonction de transfert des RNAs. Dans cette thèse une nouvelle méthode de compression dans le cas de données périodiques est présentée.

Une vecteur d'entrée est composé de 6 courants électriques de 2000 valeurs chacun. Comme il est montré sur la figure 14, un convertisseur numérique de 16 bits transforme les 2000 valeurs d'un courant en une matrice de 16x2000 valeurs. Ensuite, cette matrice est réduite en un vecteur ligne de 16 valeurs grâce à une moyenne sur les colonnes de la matrice. Cette méthode de compression est appliquée sur les 6 courants pour donner un vecteur d'entrée de 96 valeurs (au lieu de 12,000 valeurs).

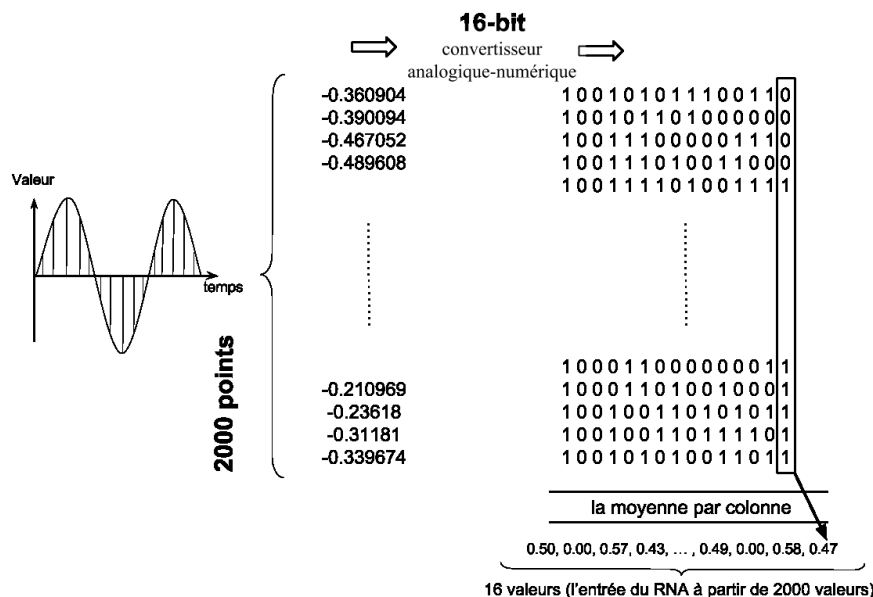


FIGURE 14 – Compression des vecteurs d'entrée du RNA.

Après la réalisation de cette compression numérique, nous proposons de commencer les tests pour différencier et localiser 16 défauts différents. Ces défauts représentent des courts-circuits sur le bobinage coté stator ou rotor. La détection de ces défauts le plus tôt possible évite la destruction totale de la génératrice. Pour cela nous allons utiliser et configurer un réseau de neurones concurrent.

La principale difficulté lorsque l'on veut utiliser un RNA est sa configuration. Il existe un certain nombre de règles pour configurer les paramètres des couches (N, M, fonction d'activation, nombre de neurones dans une couche) mais elles ne reposent sur aucunes démonstrations solides garantissant le fonctionnement de la règle dans tous les cas. Afin de configurer notre RNA de type Feed-Forward, nous allons tester les configurations à partir des paramètres de la Table 1.

Identifiant	N	M	Fonction d'activation	Neurones (couche cachée)
1	0.1	0.1	tanh	20
2	0.9	0.1	tanh	20
3	0.9	0.1	sigmoid	20
4	0.9	0.1	sigmoid	50
5	0.9	0.1	sigmoid	70
6	0.9	0.1	sigmoid	60
7	0.9	0.5	sigmoid	55
8	0.9	0.2	sigmoid	54
9	0.7	0.1	sigmoid	56
10	0.9	0.1	sigmoid	56
11	0.9	0.5	sigmoid	56

TABLE 1 – Paramètres testés pour la configuration du RNA.

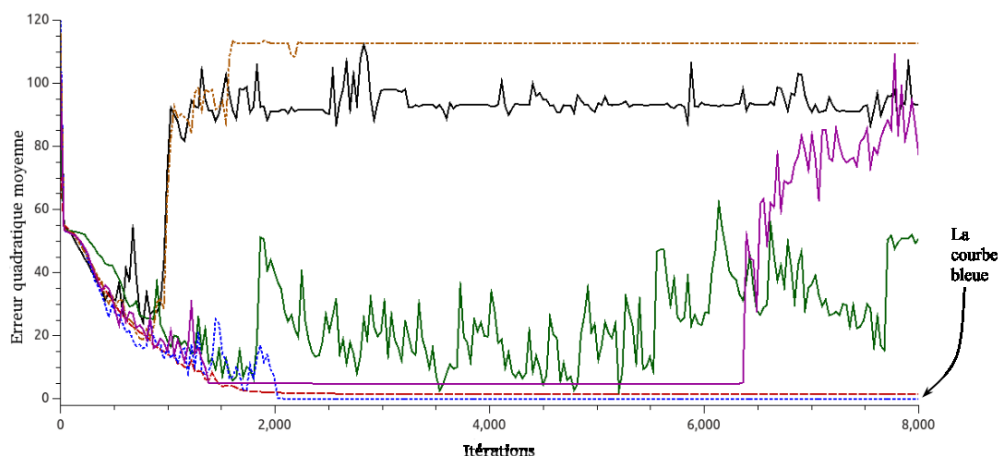


FIGURE 15 – Erreur quadratique moyenne des simulations concurrentes pour la configuration du RNA.

La Table 1 présente les paramètres de configuration présentés sont utilisés pour différencier 12 fautes différentes, 6 du coté rotor (F1-F6) et 6 du coté stator (F7-F12). La figure 15 montre les erreurs quadratique moyenne issues de l'apprentissage du RNA par l'exécution des simulations concurrentes. Les résultats indiquent qu'il y a une convergence de certaines configurations, la courbe en bleue est celle qui converge le mieux en la comparant avec les autres configurations..

Afin de nous assurer des effets de la compression sur l'erreur d'apprentissage du RNA avec les paramètres de l'expérience 9 (courbe en bleue, figure 15), la figure 16 trace les deux erreurs obtenues avec et sans compression. Cette figure montre que la compression est indispensable à la convergence du RNA.

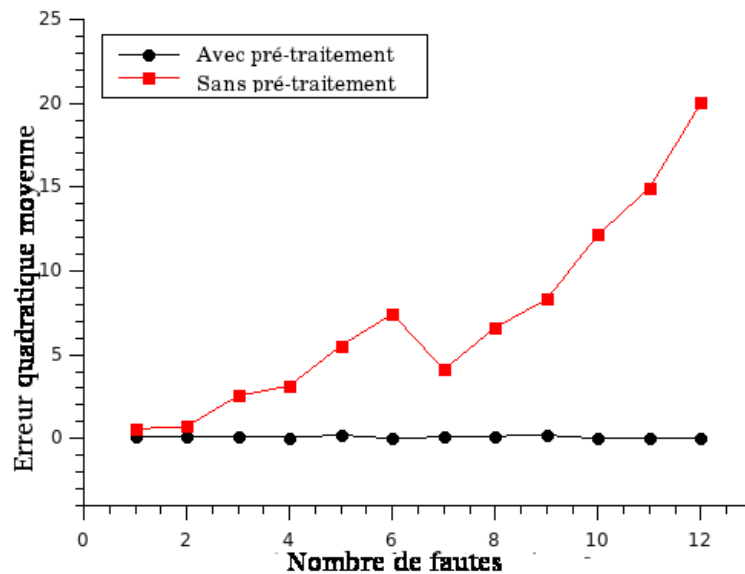


FIGURE 16 – Erreur quadratique avec et sans compression des données en ajoutant .

Optimisation avec un RNA multi-étages

Lorsqu'on désire utiliser le RNA précédent avec un nombre de fautes supérieures à 12, la convergence n'est plus respectée du fait de l'augmentation de la complexité du système principalement due à la ressemblance des défauts diagnostiqués. Afin de pouvoir identifier 5 fautes électriques supplémentaires (F13 - F16), nous avons fait évoluer notre approche en utilisant un RNA multi-étages. Cette architecture multi-étages est construite à partir d'une analyse statistique des données et conduit à décomposer le problème en sous problème facilement modélisable par des étages de réseaux de neurones communiquant. Cette étude statistique est basée sur le calcul de la distance

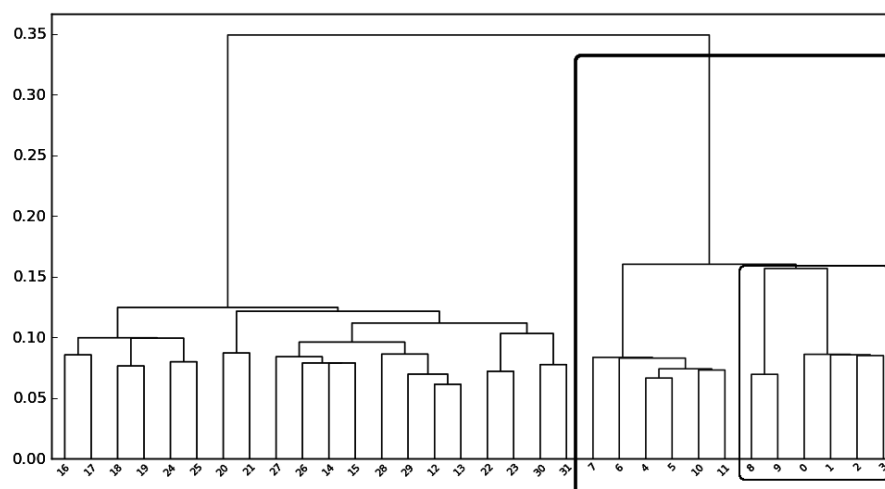


FIGURE 17 – Dendrogram des vecteurs d'entrée.

Euclidien des vecteurs d'entrée qui permet d'identifier les défauts qui se ressemblent. Un Dendrogramme est un graphique arborescent qui présente les ressemblances entre les vecteurs d'entrée. La figure 17, représente un Dendrogramme de 32 vecteurs d'entrée différent qui représente les 16 fautes électriques causées par des courts-circuits du bobinage des deux côtés rotorique et statorique. L'étude de ce Dendrogramme nous amène à diviser les tâches réalisées par les réseaux de neurones sur différents niveaux comme présentée sur la figure 18.

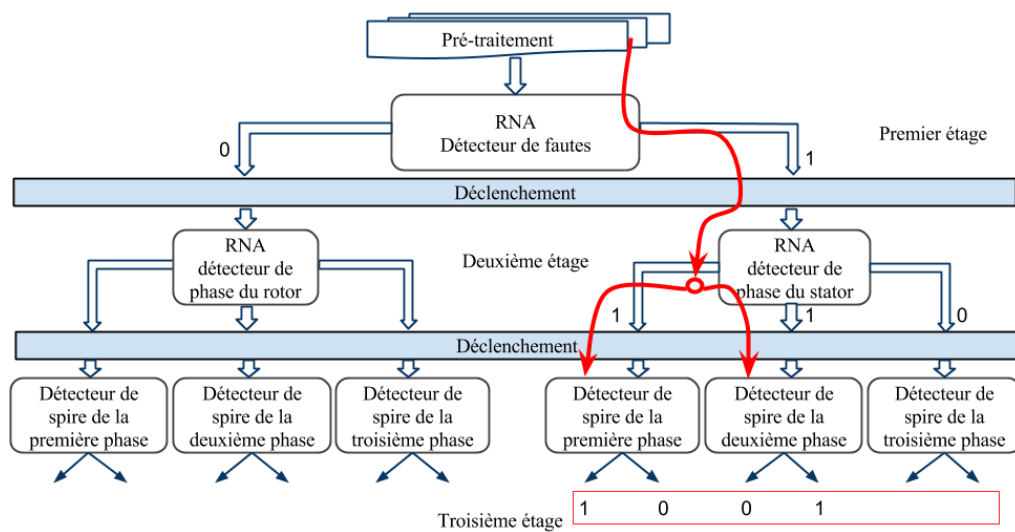


FIGURE 18 – RNA avec une architecture à multi-étages.

Le premier étage est capable de détecter si l'état de la machine est sain, fautif du côté statorique ou fautif du côté rotorique. Le deuxième étage est responsable de l'identification des défauts sur le niveau des phases. Le troisième étage est responsable de la localisation des défauts au niveau des spires sur une phase précise. D'après la figure 18, le tracé en rouge est le résultat de l'identification d'une faute coté stator, sur la première spire de la première phase et sur la deuxième spire de la deuxième phase.

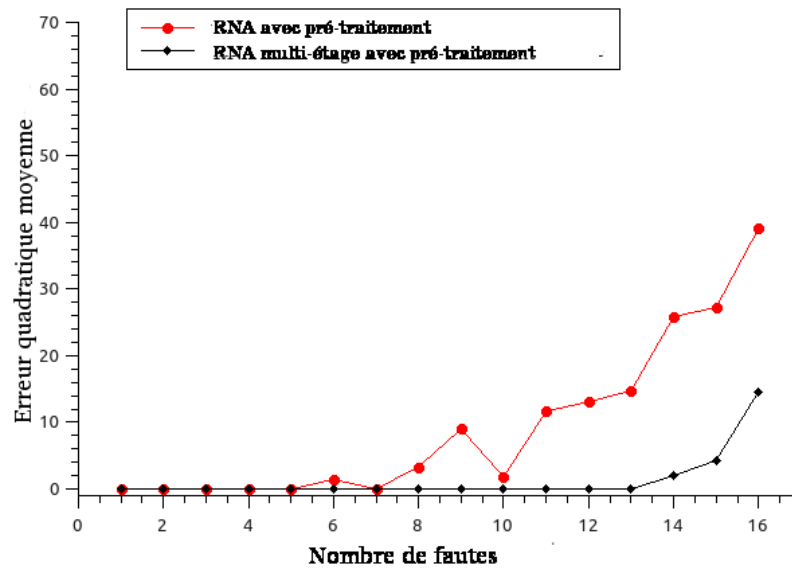


FIGURE 19 – Erreurs quadratiques d'apprentissage entre l'architecture multi-étages et classique.

La figure 19 présente une comparaison de l'erreur quadratique moyenne d'apprentissage entre le premier RNA simple et le second RNA multi-étages. Dans les deux cas, la compression des données en entrée est réalisée et les courbes montrent bien que la capacité de localisation a augmentée avec la solution basée sur le RNA multi-étages.

D'un point de vue de l'implantation matériel de la solution proposée, nous pouvons penser que du fait d'une part de la simplicité de l'algorithme de compression (opération de conversion analogique/numérique, opérations mathématique classiques comme l'addition) et d'autre part de la simplicité d'un réseaux de neurones (opérations mathématiques sur des valeurs réelles représentant les poids des neurones), il sera facile d'implanter notre approche sur un système embarqué comme un FPGA pour valider notre solution sur un banc de test.

Conclusion

Les travaux présentés dans cette thèse concernent l'association de la modélisation et de la simulation à événements discrets avec la technique des réseaux de neurones pour modéliser des systèmes complexes comme les machines électriques. Le formalisme DEVS reconnue pour sa modularité et sa hiérarchie de description, a été utilisé pour proposer une modélisation des réseaux de neurones de type Feed-Forward. DEVS a

permet un découpage proche de celui utilisé pour décrire les réseaux de neurones. Une librairie de modèles atomiques à été proposée pour modéliser les différentes couches d'un réseau de neurones et pour modéliser de manière séparé les modèles atomiques responsables de l'apprentissage du réseaux de neurones. La librairie a été implémentée dans l'environnement de modélisation DEVSimPy offrant ainsi aux utilisateurs du logiciel la possibilité de modéliser des systèmes basés réseaux de neurones. Tous les utilisateurs de réseaux de neurones savent qu'il n'existe aucunes règles fiables pour configurer un réseaux de neurones. La configuration s'effectue en testant plusieurs paramètres relatifs aux réseaux de neurones (momentum, type de fonction d'activation, nombre de neurones dans une couche, etc.). Il est donc nécessaire de proposer une solution pour l'aide à la configuration des réseaux de neurones dans DEVS. A partir de BFS-DEVS (Behavioral Fault Simulator for DEVS), nous avons généralisé et amélioré l'intégration des algorithmes de la simulation comparative et concurrente dans le formalisme DEVS. En s'appuyant sur la définition d'une fonction concurrente qui doit être associée aux fonctions de transition d'un modèle atomique et grâce au concept de la programmation orienté aspect, nous avons proposé une solution générique pour définir et manipuler des simulations concurrentes dans le formalisme DEVS. L'implémentation de ces travaux a été également réalisée dans l'environnement DEVSimPy par le biais de la programmation orientée objet et aspect et des patrons de conception UML comme le Singleton. Ces notions ont permis d'implémenter une extension dans DEVSimPy destinée à la gestion des simulations concurrentes sans affecter les algorithmes du formalisme DEVS. La mise en application de ces travaux a été réalisée dans le cadre de la modélisation et la simulation des défauts dans les machines électriques afin de réaliser leur diagnostic. Nous avons proposé une approche innovante reposant sur la modélisation d'une génératrice à induction à rotor bobiné par un réseau de neurones DEVS multi-étages configuré grâce à des simulations comparatives et concurrentes dont les entrées numériques on été compressées. Les résultats montrent que le diagnostic à l'aide de ce type de réseaux de neurones permet d'identifier 17 fautes électriques (de type court-circuit) de nature différentes et ceux avec des taux de convergence raisonnables. Les simulations ont été réalisées à partir de signaux numérisés provenant d'un banc de test réel. L'approche présentée dans cette thèse peut être rapidement implantée dans des systèmes embarqués comme

des FPGA et nous pouvons penser qu'un prototype pourrait valider notre approche sur un banc de test réel.

Contents

Abstract	v
Acknowledgments	vii
Résumé	x
List of Figures	xli
List of Tables	xlvi
List of Algorithms	xlvi
Abriviations	xlvii
General Introduction	1
Research Topics and Objectives	1
Methodological Design	4
Outline	7
1 State of the Art	9
1.1 Electrical Machines Diagnosis	9
1.1.1 Need for Condition Monitoring and Fault Detection	11
1.1.2 Faults in Wound-Rotor Induction Machines	12
1.1.3 Diagnostics Techniques Overview	15
1.1.4 Summary	17
1.2 Artificial Neural Networks (ANNs)	18
1.2.1 History	18
1.2.2 Feed-Forward Architecture	19

1.2.3	Learning Algorithms	22
1.2.4	Data Pre-Processing	26
1.2.5	Application Domain Overview	27
1.2.6	Summary	29
1.3	Discrete Event System Specification Formalism	30
1.3.1	DEVS Modeling	30
1.3.2	DEVS Simulation	34
1.3.3	DEVSImPy Environment	35
1.3.4	Summary	38
1.4	Comparative and Concurrent Simulation (CCS)	39
1.4.1	CCS Concepts	39
1.4.2	Example on Digital Circuits	40
1.4.3	Simulation Signature	41
1.4.4	CCS Properties and Advantages	42
1.4.5	DEVS/CCS Related Works	43
1.4.6	Summary	44
1.5	Conclusion	46
2	DEVS-Based ANN	47
2.1	Introduction	47
2.2	ANN/DEVS Mapping Study	48
2.2.1	Neuron Architecture Level	49
2.2.2	Layer Architecture Level	50
2.2.3	Layer Level with Reduced Number of Messages	52
2.2.4	Comparison and Selection	53
2.2.5	Summary	54
2.3	ANN/DEVS Modeling	55
2.3.1	DEVS-Based ANN Design	55
2.3.2	Feed-Forward Calculations Model Set	57
2.3.3	Back-Propagation Learning Model Set	60
2.4	ANN/DEVS Implementation with DEVSImPy	65
2.5	Conclusion	68

3	DEVS-Based CCS	69
3.1	Introduction	69
3.2	DEVS-Based Concurrent Simulation	73
3.2.1	The Concurrent Behavioral Function Modeling	73
3.2.2	Singleton Signature Database	76
3.2.3	The Concurrent Behavioral Function Simulation	78
3.3	DEVSImPy Implementation	81
3.3.1	Classes Description	81
3.3.2	DEVSImPy Global Plug-in Integration Functions	84
3.3.3	Modeling and Simulation Sequence Diagram	87
3.3.4	Validation: PC Model	88
3.4	Conclusion	91
4	Concurrent DEVS-based ANN	92
4.1	Introduction	92
4.2	Adding the Concurrent Behavior to DEVS	93
4.3	ANN Modular Optimization for Classification	98
4.3.1	Introduction	98
4.3.2	Pattern Analysis	99
4.3.3	Multistage ANN for Classification	101
4.3.4	Conclusion	103
5	Case of Study: Electrical Machine Diagnosis	104
5.1	Introduction	104
5.2	Short-Circuit Diagnosis	106
5.2.1	Set-up Description	106
5.2.2	Learning and Test Patterns Study	109
5.3	Data Pre-Processing	111
5.4	ANN Configuration and Simulation	116
5.5	Multistage Optimization	121
5.5.1	Patterns Analysis	122
5.5.2	Multistage Architecture	123
5.6	Conclusion	127

Table of Contents

General Conclusion and Perspectives	128
6 Annexes	134
List of Publication	146
Bibliography	147
Abstract	157

List of Figures

1	L'architecture d'un neurone artificiel j.	xv
2	Modélisation DEVS d'un RNA.	xvi
3	Modélisation DEVS d'un RNA avec la phases d'apprentissage.	xvii
4	Un réseau de neurones implémenté dans DEVSIMPy.	xvii
5	Coresspondance de la simulation comparative et concurrente avec DEVS.	xix
6	La simulation du comportement concurrent d'un modèle atomique DEVS.	xxi
7	Diagramme de classes UML pour l'implémentation de la SCC dans DEVSIMPy.	xxii
8	Interfaces de l'extension global pour la configuration du comportement concurrent.	xxiii
9	Diagramme de séquences de la simualtion comparative et concurrente dans DEVS.	xxiv
10	Stratégie du travail réalisé.	xxv
11	Banc de test.	xxvi
12	Échantillons des trois courants statoriques en régime permanent.	xxvi
13	Préparation des données avant et après le réseau de neurones.	xxviii
14	Compression des vecteurs d'entrée du RNA.	xxix
15	Erreur quadratique moyenne des simulations concurrentes pour la configuration du RNA.	xxx
16	Erreur quadratique avec et sans compression des données en ajoutant	xxxi
17	Dendrogram des vecteurs d'entrée.	xxxi
18	RNA avec une architecture à multi-étages.	xxxii
19	Erreurs quadratiques d'apprentissage entre l'architecture multi-étages et classique.	xxxiii
20	Methodological design.	5

1.1	The two main electric components of an electrical three-phase induction machine.	10
1.2	Fault diagnosis process.	12
1.3	Topology of the different short-circuits in both stator and rotor windings: each stator phase is (1:NS-2:1) turns and each rotor phase is (1:NR-2:1).	13
1.4	Neuron Perceptron Architecture.	19
1.5	Most common activation functions in MLP neural networks.	20
1.6	Example of multiple layer feed-forward neural network with data flow direction.	21
1.7	Feedback learning process.	23
1.8	Learning with back-propagation algorithm.	24
1.9	Pre-processing for artificial neural network.	26
1.10	Separability of the different spaces.	28
1.11	The DEVS atomic model in action.	31
1.12	Atomic model state trajectory.	32
1.13	Couplings in coupled models.	33
1.14	Example of modeling and simulation of DEVS models.	35
1.15	DEVSIMPy general interface.	36
1.16	Atomic model properties panel.	37
1.17	DEVSIMPy plug-in manager.	37
1.18	General example of a comparative and concurrent simulations.	39
1.19	Logic gate fault propagation effect.	40
1.20	General example of comparative et concurrent simulation.	41
1.21	The BFS-DEVS simulation kernel positioning.	44
2.1	Design of one neuron modeling level.	49
2.2	One atomic model per layer modeling approach.	51
2.3	Reduced messaging approach.	52
2.4	Messaging comparison between the three proposed approaches.	54
2.5	DEVS-based neural network modeling with the feed-forward architecture and the additional training layer.	56
2.6	The non-calculation (<i>Input</i>) layer DEVS atomic model.	57

2.7	Block diagram for <i>Input</i> layer DEVS atomic model (Non-calculation layer model).	58
2.8	Calculation layer DEVS atomic model.	59
2.9	Block diagram for calculation layer DEVS atomic model.	60
2.10	Error-Generator DEVS atomic model	61
2.11	Delta-Weight DEVS atomic model.	63
2.12	The DEVS state diagram of the Delta-weight DEVS atomic model.	63
2.13	The DEVS-based ANN with DEVSIMPy.	65
2.14	Configuration panels for the DEVS atomic models inside DEVSIMPy.	66
2.15	XOR quadratic error DEVSIMPy simulation results.	67
3.1	Concurrent behavior liking in the DEVS atomic model.	70
3.2	Basic entities and relations in modeling and simulation.	71
3.3	The layers between an application domain and the classic DEVS simulation core.	72
3.4	A single atomic concurrent DEVS model with a signature pointer.	76
3.5	Singleton signature database.	77
3.6	The concurrent DEVS atomic model simulation.	79
3.7	DEVS-based CCS UML class diagram.	82
3.8	DEVSIMPy plug-in manger interface.	85
3.9	Model association panel.	86
3.10	Simulation configuration panel.	86
3.11	Sequence diagram for the concurrent simulation scenario inside DEVSIMPy.	87
3.12	Concurrent PC DEVS coupled model for concurrent simulation.	89
4.1	DEVS-based ANN with a concurrent learning behavior.	93
4.2	The simulation panel interface of the ANN configurations.	97
4.3	ANN simulation results.	97
4.4	The company job hierarchy.	98
4.5	Work distribution on a multistage ANN explanation example.	100
4.6	Dendrogram cluster analysis on standardized Euclidean distance on random data.	100

5.1	Set-up for different tests.	107
5.2	Sample of 3 normalized stator currents (2000 samples each) at steady state with fault F10.	107
5.3	Data pre/post processing for neural networks.	111
5.4	Digital compression procedure.	113
5.5	Mean squared learning error for the concurrent simulation presented in Table 5.1.	117
5.6	Learning and test error comparison between compressed and non-compressed data.	119
5.7	Dendrogram for similarity study between 32 input vectors.	122
5.8	Proposed multistage neural network for WRIG short circuit classification.	123
5.9	Absolute error, comparing the performance of a single ANN to the multistage architecture.	125
5.10	Global view of the realized work.	131

List of Tables

1	Paramètres testés pour la configuration du RNA.	xxx
2.1	Set of teaching vectors of XOR function.	66
3.1	The input and output data of the concurrent PC system.	90
4.1	The truth table of the 4-Bit parity problem.	96
4.2	ANN test configurations.	96
5.1	ANN configurations.	116
5.2	Results for classification of the first 12 faults with a 16-bit AD (simulation ID = 9).	118
5.3	Influence of the AD converter accuracy.	120
5.4	Iteration number needed to get the best results during the learning mode.	126

List of Algorithms

0.1	La fonction du comportement concurrent de la couche d'entrée d'un RNA.	xxv
1.1	Back-propagation algorithm steps.	25
3.1	Concurrent decoration function.	86
3.2	Concurrent behavioral function for the CPU atomic model.	89
4.1	Concurrent behavioral function for the Input atomic model.	94

Abriviations

ANNs	Artificial neural networks
AOP	Aspect-oriented programming
BFS	Behavioral fault simulation
BP	Back-propagation learning algorithm
CCS	Concurrent and comparative simulation
CFS	Concurrent fault simulation
DEVS	Discrete event system specification
DSP	Digital signal processing
FF-ANN	Feed-forward artificial neural network
FPGA	Field-programmable gate array
GUI	Graphical user interface
LSBs	Least significant bits
MLP	Multi-layer perceptron
MSBs	Most significant bits
OOP	Object-oriented programming
SPE	Science pour l'environnement
VHDL	VHSIC hardware description language
WRIGs	Wound rotor induction generators

WRIMs Wound rotor induction machines

General Introduction

The core work presented in this thesis deals with electrical machine fault diagnosis by the direct usage of experimental digital data through an artificial neural network modeling with a discrete event approach. The main focus of this work will be to provide the possibility to re-design artificial neural networks (ANNs) to be able to compare different configurations and performances for the electrical faults diagnosis. Moreover, the work involves an investigation based on concurrent & comparative simulation (CCS) concepts and its implementation in the discrete event system specification (DEVS) formalism. The research topics, objectives, methodological design and the document outline are pointed in this general introduction.

Research Topics and Objectives

The technique for condition monitoring and fault detection of electric machines has been developed over the last 80 years, starting from human analysis and leading up to modern decision processes. For the last 20 years, the squirrel-cage three-phase induction machine has been under focus, which is without a doubt, the most interesting machine in terms of cost and reliability. Many research works are based on diagnostic techniques around induction machines, mostly involving electrical faults. In modern wind farms, a large majority of generators are based on wound rotor induction machines (WRIMs) with a low number of poles and a planetary gearbox to adapt machine rotor shaft speeds to the blade speeds. Researchers have put more emphasis on investigating control of wound rotor induction generators (WRIGs) and less on fault detection and localization. The basic idea of this work is to make diagnostic techniques more economical and more reliable for use at a large scale. This helps to provide predictive maintenance to modern wind farms which is crucial to an efficient

operation. It is well known that any AC (alternative current) WRIM can still operate with shorted turns even at its rated load [1]. It is clear that early stage detection is based on the need for predictive maintenance to avoid full winding failure due to thermal propagation in the machine windings. In this thesis, short-circuit tests are almost at their early stage with one or two shorted turns in each phase and on each side (stator or rotor).

The different electrical faults characterized by signal processing techniques have been qualified as complex and it is always difficult to obtain a clear conclusion on any fault associated to time, frequency, or time-frequency analysis [2]. It is clear that frequency domain-based fault detection has been dominant for the last twenty years but it has always faced two problems: (i) data collection length, (ii) computational burden. In this way, decision techniques, mainly based on artificial intelligence (AI), have been used to help the maintenance process on any type of electrical machine. Artificial neural networks (ANNs) have been used with success to help with decision processes in order to inform the predictive maintenance schedule [2,9]. More recently, ANNs have substituted the data processing and computational techniques in order to mix both classification and decision processes. The data collection has to come from a minimum number of sensors being placed around electrical machines to reduce monitoring costs.

Nowadays, many sensors have digital outputs which can be connected to any system such as a computer, a digital signal processor (DSP), or a field-programmable gate array (FPGA) board to perform complex operations directly without any signal conversion, avoiding information loss. This will help the implementation of new diagnostic tools/techniques and help to increase the use of computer-based artificial intelligence. It is important to note that after avoiding the frequency domain diagnostics problems, ANNs will face other problems: (i) the need to prepare the input data (pre-processing), (ii) it needs to be trained before anyone can use it for any purpose, (iii) the choice of the neural architecture, the training algorithm, and the configuration parameters.

The artificial intelligence domain grows every day with new algorithms and new architectures. ANNs have become a very interesting domain since the eighties when the back-propagation learning algorithm and the feed-forward architecture were first

introduced [17,21]. As time passed, ANNs were able to solve non-linear problems, and were being used in classification, prediction, and representation of complex systems. Nowadays, ANNs are still in the research domain to enhance performance and facilitate the configuration parameters needed for the learning process [18,22–25]. Comparing multiple configuration parameters and learning algorithms is very common for ANN users and algorithm developers. The use of the Comparative and Concurrent Simulation (CCS) can be a practical solution for ANN users and developers to use, test and compare different learning algorithms and configurations. [26] is a practical solution.

The objective of this work is to define fault detection and localization for wound rotor induction generators by using new techniques to pre-process data and enhance neural network simulation. This work has been integrated and implemented with the discrete event specification (DEVS) formalism introduced by Pr. B.P. Zeigler in the '70s [14]. DEVS separates modeling and simulation in a hierarchical way that can be described with a state transition table and continuous state systems. DEVS formalism is a very interesting platform which can be adapted for multiple applications by adding extensions to it.

The concurrent and comparative simulation is the concept of simulating multiple experiences within a single simulation execution with a concurrent behavior comparing the simulation progress for each step. The CCS is a general concept that enables the comparison between experiments with different paths and data values within a single simulation [26,27]. One of the first applications of the CCS is concurrent fault simulation (CFS) to simulate faults (mainly for digital systems) [16]. CCS can be applied to many fields and it matches the idea of comparing multiple ANN configurations during the learning phase. This idea enables the simulation of multiple ANN configurations and compares their performances. In order to ensure a good performance, CCS will be incorporated into the formalism DEVS inside the DEVSimPy environment to insure a better training of ANNs dedicated to WRIM short-circuit diagnosis.

Methodological Design

To achieve the objective based on the motivation described in the previous section, the methodological design is detailed as follows:

1. **Study of electrical induction machines** and their faults diagnostic techniques.
2. **Study of discrete event system specification (DEVS)** and the DEVSimPy environment.
3. **Study of the comparative and concurrent simulation (CCS)** concepts.
4. **Study of artificial neural networks (ANNs)** with different architectures and algorithms dedicated to classification.
5. **Modeling of ANNs with DEVS** formalism using the DEVSimPy environment.
6. **The design of the DEVS-based CCS**, and the atomic model mathematical description.
7. **The implementation of the DEVS-based CCS** inside DEVSimPy environment and the validation of the new concurrent DEVS-based ANN.
8. **The concurrent DEVS-based ANN** simulation for electrical machine faults diagnosis and condition monitoring.

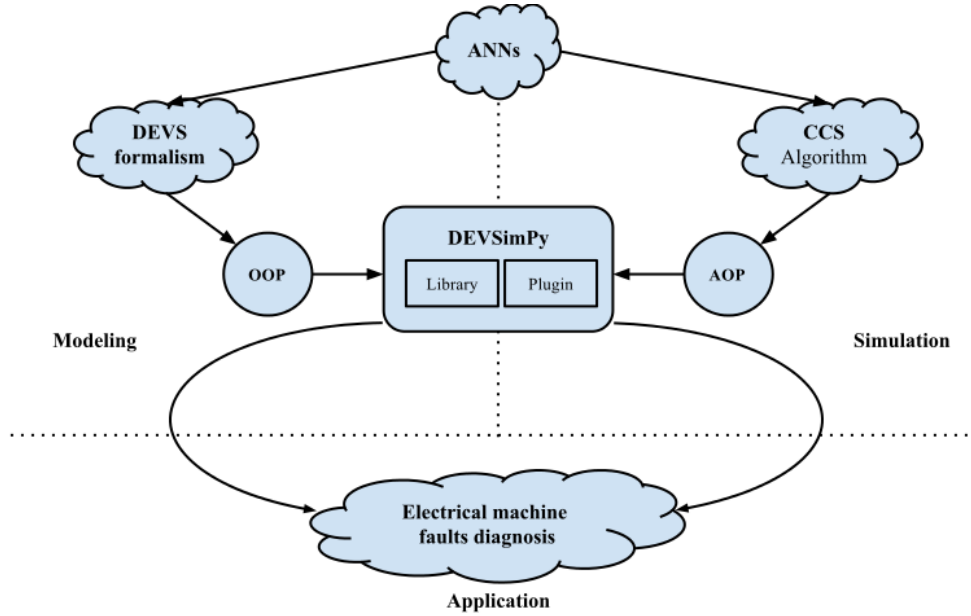


Figure 20 – Methodological design.

As shown in Figure 20 there are three different aspects of the work - modeling, simulation and application. A redesign of the artificial neural network **modeling** is proposed using the modularity of the DEVS formalism. DEVS has an object-oriented programming orientation (OOP) which means that models are represented in a hierarchical fashion. This OOP representation of the ANN is in the form of a library inside the DEVSIMPy environment.

The **simulation** is based on the proposed implementation technique of the CCS in order to deliver a concurrent behavior to DEVS without any change in the simulator. The concurrent ANN configurations and comparisons take place under the aspect-oriented programming (AOP) inside DEVSIMPy. The **implementation** of the CCS is in the form of a plug-in inside the DEVSIMPy environment in order to keep the core of the DEVS simulator unchanged. In Figure 20, DEVSIMPy is at the intersection between the simulation and modeling approaches. DEVSIMPy also provides: (i) modeling via a graphical user interface (GUI), (ii) offers the option to create a dedicated library, (iii) it proposes the implementation of plug-ins which will help the realization of the CCS.

After the validation of the concurrent ANN concept, it will be applied to the winding rotor induction machine (WRIM) faults diagnosis. The application can be

seen from two different views. On one side there is the modeling and implementation of a pre-processing technique and how to deal with the direct digital data output from the WRIM sensors for machine diagnosis. On the other side the modeling and simulation of the ANN using DEVS and the CCS implementation inside DEVSimPy using the OOP and the AOP.

Outline

This thesis is organized in five chapters. The first chapter is named **State of the Art**. This chapter contains the terms and the concept used to achieve this thesis objectives. The first part is dedicated to the induction machine diagnosis, where the importance of the early diagnosis for electrical machines and some of the technique used to date are presented. The second part explains in some details the artificial neural network, one of the widely used artificial intelligence technique for machine diagnosis. The third part is composed of a description of the DEVS formalism and the main modeling and simulation concepts. The last part of this chapter describes the comparative and concurrent simulations concept. The CCS signature is an important part of the CCS description which is widely used for the integration with the DEVS formalism.

The second chapter, named **DEVS-Based Artificial Neural Network**, presents the concept of fragmenting the ANN into several atomic models easier to use and replace. This DEVS design also can benefit from the DEVS extensions where more functionalities can be added to the formalism. An implementation is proposed inside the DEVSImPy environment where it can bring a user friendly interface.

The third chapter is named as **DEVS-Based Comparative and Concurrent Simulation (CCS)**. This chapter explains a new DEVS concurrent simulation without any changes to the classic DEVS simulator. This concept comes transparent to the simulator offering a better compatibility with other DEVS extensions.

The fourth chapter, named as **Concurrent DEVS-Based ANN**, is the place where all of the DEVS-based CCS and the DEVS-based ANN regroup together to produce a final product. Also an optimization for classification is proposed based on a statistical approach using the Euclidean distance between the input vectors of the neural network.

The fifth chapter is the **Case of Study** of the WRIM diagnosis. The data used in this chapter represents seventeen different electrical faults (winding short-circuit) on both rotor and stator sides. The first part of this chapter presents a customization of the DEVS-based ANN for the machine diagnosis. The second part unveils a new compression technique for large and periodic signals based on digital data. The third part presents the simulation results and comparison between compressed and

uncompressed data. Last is an optimization of the DEVS-based concurrent ANN and consists of proposing a multistage architecture based on statistical data analysis.

The thesis ends with a general conclusion of the presented work as well as some research perspectives on the future work.

Chapter 1

State of the Art

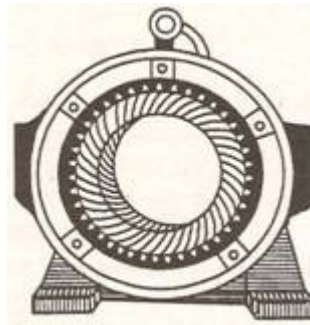
In this chapter, a review of the three domains exposed in this thesis will be detailed. As presented in the introduction the three domains are: (i) the induction machines fault diagnosis, (ii) Artificial Intelligence (AI), and more specifically Artificial Neural Networks (ANNs), (iii) modeling and simulation with the discrete event formalism DEVS and the decoration with the comparative and concurrent simulation approach that will bring a new perspective to DEVS simulation.

1.1 Electrical Machines Diagnosis

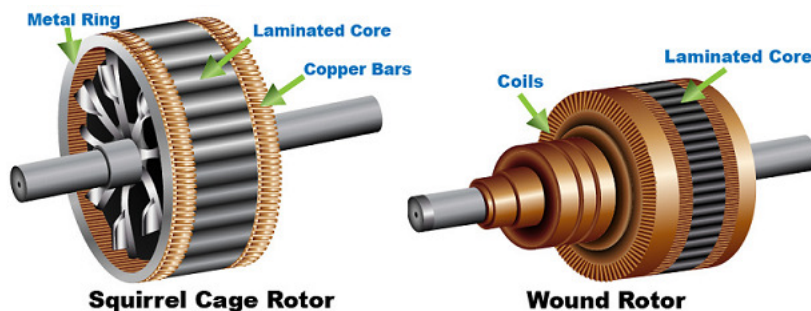
Wind energy has become one of the most important renewable energy sources and it has a great share in total power generation all over the world. In fact, the attempt to transform wind energy to electricity began in 1887-1888, when the first automatically operating wind turbine generator was built by Charles F. Brush [28]. By that time, low price fossil fuels made wind energy economically unattractive. It was only in the '70s with the oil crisis that the research turned to making low-price turbines, which are composed of a small turbine, a generator, a gearbox and a simple mechanical control method. The induction generator is a proper choice for this system since it is one of the most economical in term of implementation. The price became reasonable with these low-cost and small-sized components, even for individuals to purchase. During the 1980s, the generators were rated at 10 to 65 kW, then up 200kW. Today, wind energy developers are installing turbines rated at 200kW to 2MW [29]. According to the American Wind Energy Association, today's large wind turbines produce about 120

times more electricity than early turbine designs. In the last 25 years, five generations of wind turbine systems have been developed [5]. These generations are different based on the usage of types of wind turbine rotors, generators, control methods and power electronic converters.

In fact, electrical machines are extensively used in engineering systems. The induction generators are mostly used in wind turbines as they can be operated at variable speeds, unlike synchronous generators. An induction machine is composed of magnetic circuits which are composed of two induction circuits, rotating in respect to each other with the power transferred from one circuit to the other by electromagnetic induction. Two circuits are called rotor (the rotating element) and stator (the stationary component). In fact, the machine can be used as a motor or generator. It is an electromechanical energy conversion device that converts energy from electrical to mechanical form and vice-versa. A three-phase induction machine has one stator and one rotor which are separated by a small air-gap.



(a) Winding stator for a three-phase induction machine [30].



(b) Squirrel cage and wound rotor details [31].

Figure 1.1 – The two main electric components of an electrical three-phase induction machine.

Stator: This part of the machine is built with thin steel lamination stacked together and held in the stator housing (Figure 1.1a). The conductors making up the coils in the stator windings are looped through slots in the stator lamination. Coils in this machine are insulated from the lamination using plastic sheets and held together with string and paper to separate coil groups. The stator coils and lamination are then dipped in insulation varnish and baked to provide mechanical integrity.

Rotor: It is the rotating element in the machine, and it can be one of two types: (i) squirrel-cage or (ii) wound (Figure 1.1b). The cage rotor machines are very popular motors since they are very interesting in term of cost and reliability [32]. In this case, the rotor bars are permanently short-circuited by the end-rings, and it is not possible to add any external resistance. Standard squirrel cage rotors have no insulation since bars carry large currents at low voltages. The second type is the wound-rotor, which gives the name of WRIM. In modern wind farms, a large majority of generators are based on (WRIG) with a low number of poles and a group of gearboxes to adapt the machine rotor shaft speed to the blades speed. The wound rotor consists of a laminated cylindrical core and carries a 3-phase winding, similar to the one on the stator. The terminals of the rotor winding are connected to three insulated slip rings mounted on the rotor shaft. The rings are connected to the outside of the machine by three brushes.

1.1.1 Need for Condition Monitoring and Fault Detection

Electrical machines are frequently exposed to non-ideal and disturbed environments. That can include overload, insufficient lubrication, inadequate cooling and more. Under these conditions, electrical machines are under undesirable stresses, which put them under risk of faults or failures. The literature on condition monitoring of electrical machines is growing rapidly. Recently many review papers have been published to give state of the art for diagnostics techniques around induction machines mostly related to electrical faults [1–4].

Continuous evaluation of the health for equipments is important to detect early faults. Figure 1.2 shows the four common steps for machine monitoring, fault diagnosis and decision making. The first step for fault diagnosis is data acquisition, which

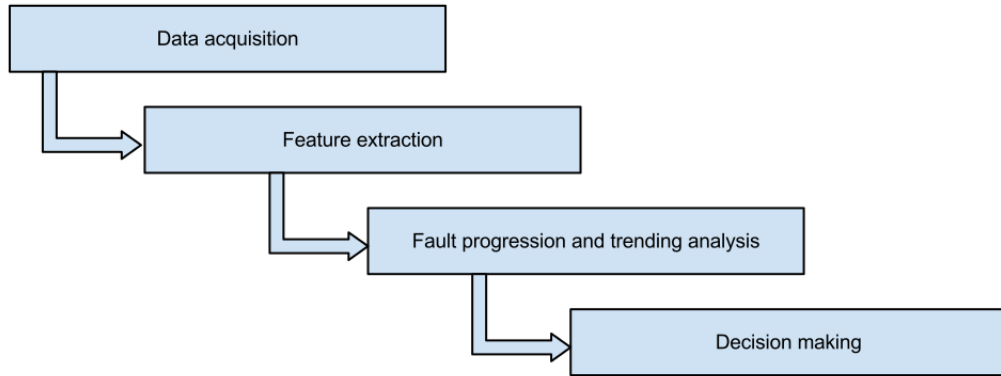


Figure 1.2 – Fault diagnosis process.

can be performed through invasive or non-invasive methods. Feature extraction is responsible for data transformation and preparation to be used for fault analysis. The feature extraction is very important for ANN as it needs data for a learning process before any use. The fault diagnosis and localization of a specific fault that has occurred in a system. Condition monitoring and early fault detection can reduce the cost of operation by: (i) reducing maintenance costs, (ii) improving failure prediction accuracy, (iii) predicting equipment failure, (iv) improving the equipment and component reliability [33].

It is important to know that a variety of faults occur within the three-phase induction machines during normal operation. In fact, early fault detection can prevent a catastrophic machine failure. Condition monitoring and fault diagnosis can be performed by several methods: (i) thermal monitoring, (ii) noise monitoring, (iii) vibration monitoring, (iv) electrical monitoring. Nowadays, among these different techniques, electrical monitoring is the most popular. In most electrical monitoring systems, no additional sensors are necessary to collect the data - it is a non-invasive system, which is cost effective.

1.1.2 Faults in Wound-Rotor Induction Machines

Electrical machine drives are subject to many different types of faults. Short turn winding faults, rotor faults, bearing faults, gear fault and misalignment are common internal faults of induction machines. The common internal faults can be categorized into two groups:

- Electrical faults
- Mechanical faults

Electrical faults include failure caused by winding insulation problems [34], some other faults not due to insulation but also stator and rotor core faults. Mechanical faults include bearing faults, air gap eccentricity, misalignment of the shaft and more.

This work will be more dedicated to electrical faults, especially winding short-circuits on both rotor and stator sides. Inter-turn short circuits in stator and rotor windings constitute a category of faults which is common in induction machines. Typically, short-circuits in stator/rotor windings occur between (i) turns of one phase, (ii) turns of two phases, (iii) turns of all phases [32]. Moreover, short-circuits between stator winding conductors and the stator core also occur and the same is valid for the rotor side. A study by Thomson [35] shows that 38% of machines faults are electrical stator faults, 10% are electrical rotor faults, 40% are due to bearings and 12% from other causes. The machine failure caused by the electrical problems in stator windings is considered the most common fault. In that case, machine aging means mainly that the electrical insulation system (EIS) is aging. High temperature during machine operation can cause short-term insulation degradation. On the other hand, insulation can just be deteriorated by aging.

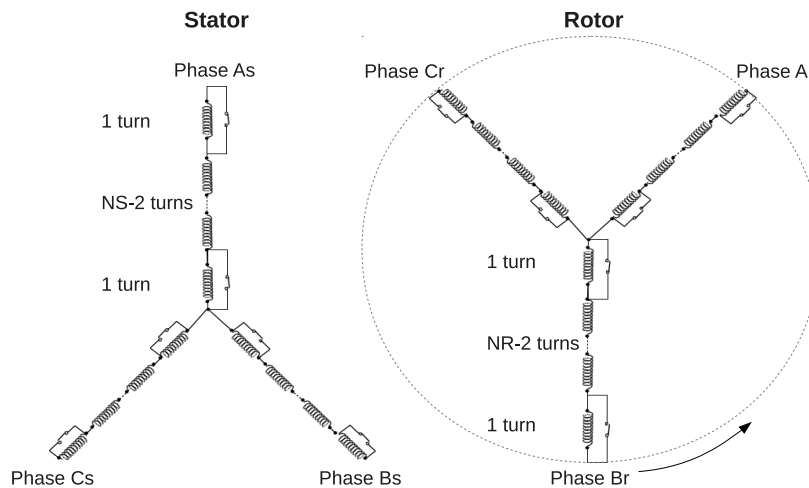


Figure 1.3 – Topology of the different short-circuits in both stator and rotor windings: each stator phase is (1:NS-2:1) turns and each rotor phase is (1:NR-2:1).

Whatever the reason is, when the insulation begins to be degraded, short-circuits inside both stator and rotor may happen. The short-circuited winding is recognized

as one of the most difficult failure to detect [3]. The usual protection might not work or the motor might keep running while heating in the shorted turns would cause critical insulation breakdown. If the shorted-turn is left undetected, it will propagate, leading to phase-ground or phase-phase faults. These types of faults can lead to irreversible damage to the core, and the machine must be removed from service [36]. In this work, the only type of electrical faults used in all experiments is based on turn-to-turn windings short-circuits on both stator and rotor sides (Figure 1.3). For experimental reasons, short-circuits less than one turn will not be considered. The following list represents the winding faults that will be studied:

For the rotor side:

- F1: short-circuit on the first two turns of the first two phases, and on the first turn of the third phase.
- F2: short-circuit on the first two turns of the three phases.
- F3: short-circuit on the first two turns of the first phase, and on the first turn of the second phase.
- F4: short-circuit on the first two turns of the first two phases.
- F5: short-circuit on the second turns of the first phase.
- F6: short-circuit on the first two turns of the first phase.

For the stator side:

- F7: short-circuit on the first turn of the first two phases.
- F8: short-circuit on the second turn of the first two phases.
- F9: short-circuit on the second turn of the first phase.
- F10: short-circuit on the second turn of the second phase.
- F11: short-circuit on the first two turns of the second phase.
- F12: short-circuit on the first two turns of the first phase.
- F13: short-circuit on the first turn of the second phase.
- F14: short-circuit on the first turn of the first phase, and the second turn of the second phase.
- F15: short-circuit on the second turn of the first phase, and the first turn of the second phase.
- F16: short-circuit on the first turn of the first phases.

1.1.3 Diagnostics Techniques Overview

Since non-invasive sensors offer a relatively simple and cost effective fault diagnosis, more research is given to electric current analysis rather than vibration or acoustic analysis in induction machines [6]. In fact, an ideal diagnostic technique should take the minimum measurements necessary from a machine and by analysis extract a diagnosis in minimum time. Non-invasive monitoring is achieved by easily measuring electrical quantities like current, voltage, flux. In any case, a key item for any fault detection is proper signal conditioning and processing. It is important to note that most interesting technique is based on electrical measurements, mainly because they are readily available in the power converter and for signal processing, but also attractive because they are non-invasive [37].

Signal processing techniques for fault diagnosis can be classified into three main classes: (i) time domain, (ii) frequency domain, (iii) time-frequency domain [36]. Spectral estimation (frequency domain) is widely used in electrical machine diagnosis and it is mainly divided into three subclasses: (i) nonparametric, (ii) parametric, (iii) high-resolution methods. State-of-the-art in diagnostics techniques for three-phase electrical machines relies on frequency analysis of stator currents [3, 4, 6]. Frequency analysis is usually computed by sampling the signal and adopting the fast Fourier transform (FFT) algorithm. Other types of frequency estimation are also based on FFT but with resolution improvement are introduced with the idea to focus on some special frequencies and not the full-length FFT. Example: the zoom-FFT (ZFFT) and the chirp Z-transform [38].

The techniques based on time-domain analysis are used for an efficient detection of mechanical imbalances. In [39], the author presents a full-time domain-based method for quantitative evaluation of electrical faults in induction machines. There are different techniques in the time-domain analysis that can be noted as amplitude and phase demodulation [40], noise cancellation [41,42], speed and torque estimations [43]. Time-domain analysis is a powerful tool for three-phase squirrel cage induction machines in faulty conditions [44].

Time-frequency analysis consists of the 3-D time, frequency and magnitude representation of a signal, which is inherently suited to indicate transient events. An important use of the time-frequency analysis is the ability to filter out a particular

frequency component using a time varying filter. The main advantage of the time-frequency analysis is discovering the patterns of frequency changes, which usually represent the nature of the signal. As this pattern is identified, the machine fault shown by this pattern can be identified.

It is important to note that signal processing allow characterizing the different electrical faults, they have been qualified as complex and it is always difficult to perform a clear conclusion on any fault associated to time, frequency or time-frequency analysis. In the supervision of electrical equipment, diagnosis systems based on conventional computing techniques have been recently replaced by new decision techniques based on artificial intelligence (AI) and especially artificial neural networks [7–9]. Traditional systems are developed as a model-based approach which is able to consider different fault conditions and to schedule a wide series of operating conditions. As a matter of fact, their implementation in a single application leads to complex programs difficult to maintain and to manage. On the other hand, the term AI includes neural networks, fuzzy logic, fuzzy-neural network, genetic algorithms and more. These techniques require an initial training which is critical for optimal performances. In fact, the training mode of a neural network can be very tricky and this problem will be studied more deeply in this work. By stating that the training phase is possible, these techniques are efficient, simple and can be adopted successfully for the diagnosis of electrical machine failures. For this purpose, it will be proposed to train the dedicated ANN directly with digital signals coming from the sensors implemented around the WRIM under condition monitoring. ANN is used for fault detection using frequency, time, and time-frequency domains for multiple types of faults. In [10] a supervised multi-layer perceptron (MLP) neural network is used for the detection of broken bars based on the use of the frequency domain analysis [11–13]. Almost two decades ago, ANN digitized tasks were imagined [45, 46] and recently they have been used for multiple-fault detection in stream turbines [47]. However, this technique has never been implemented for electrical machines fault detection nor for power systems condition monitoring.

1.1.4 Summary

The diagnosis of electrical machine has been an important research subject for more than a century. There are many types of faults inside three-phase electrical machines, but electrical faults are considered the most common ones. In this work, electrical faults, especially inter-turn short-circuit in both stator and rotor sides, are explained in depth. This is justified by the fact that WRIG in low-voltage can keep working for a while even with short-circuits, but early fault detection can prevent a focal machine failure. In the last years, more research works were focused on AI in order to improve performances of the traditional model-based methods. Artificial neural networks were commonly used along side with the traditional diagnosis time, frequency and time-frequency techniques. In this thesis the ANN will be used as a standalone technique and will be fragmented in order to enhance their modeling and simulation approaches to get easier training and learning.

1.2 Artificial Neural Networks (ANNs)

This section addresses the birth of a new artificial neural network, its architecture, learning algorithms and ideas about application domains. This section will also give the reader the ANN technical terminology used in this thesis.

Artificial neural network are used in many application in different domains. Generally, problems like patterns recognition/classification or function approximation are mostly used among these applications. The most famous ANN features is the ability of mapping between the input data and the desired output.

There are many types of neural networks. Each type has its own advantages and drawbacks, depending on the data type and the application nature. When used for classification, the choice can easily go to the feed-forward architecture, which is characterized by the fact that information flows only from input forwardly through the network to the output.

1.2.1 History

The term neural network was traditionally used to refer to the network of biological neurons. Today, it more commonly refers to artificial neural networks.

In 1943 an important article was published to talk about how neuron nets might work [48]. The neuro-physiologist Warren McCulloch and the mathematician Walter Pitts were the authors of this work. First they discussed the theory *Nets without circles* and then *Nets with circles*. These theories are based on some assumptions that the activity of the neuron is an all-or-none process. They also said: "*for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time*". In 1949, Donald Hebb wrote a paper with the title of "The Organization of Behavior" [49], which pointed out that each time the neural pathways are used they get stronger and the connection between neurons is enhanced. This concept is fundamental for human learning. At the end of the fifties, as computer became more advanced, Bernard Widrow and Marcian Hoff developed models called Multiple ADaptive LINear Elements (MADALINE). These were firstly used as adaptive filters which eliminated echoes on phone lines. This was the first neural network to be applied to a real world problem. This perceptron

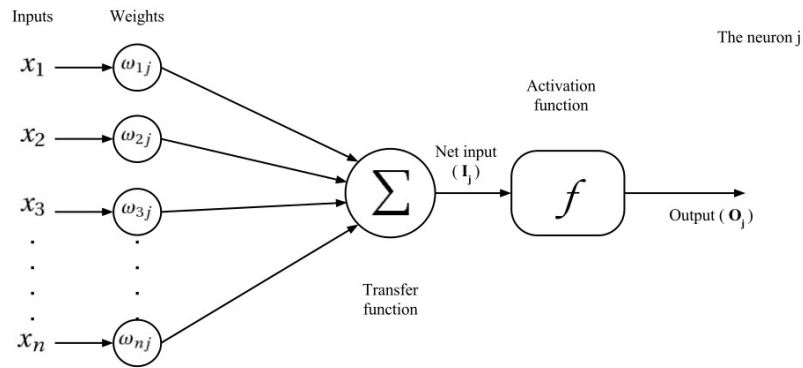


Figure 1.4 – Neuron Perceptron Architecture.

was not able to solve problems like the XOR (exclusive or). Such drawback led to slow down its use and development.

In the 1980s higher level programming and code generators became popular. Since then, researchers worked on how to extend the Widrow-Hoff MADALINE model to multiple layers. At this time, both back-propagation networks and Boltzmann machines were born. Since the eighties, ANNs keep improving, with different architectures and learning algorithms.

Nowadays there are many architectures and learning algorithms. This thesis will be focused on supervised learning (back-propagation algorithm) and the feed-forward (FF) architecture.

1.2.2 Feed-Forward Architecture

All neural networks are composed of interconnected computing units. These computing units are also called neurons. As Donald Hebb pointed out, that each time a neural pathway is used it gets stronger; in artificial neural networks this can be represented by the multiplication of a coefficient that can magnify or decrease the strength of this pathway. This strategy leads to a mathematical representation of the artificial neuron (Figure 1.4).

PERCEPTRON ARTIFICIAL NEURON: For every neuron j placed in layer L there is n inputs represented by x_i where $i = 1, 2, \dots, n$ (Figure 1.4). For every input signal there is a multiplication by a coefficient (weight w_i) that assure the signal strength of each input. The transfer function (Equation 1.1, Figure 1.4) is the operation of multiplication of inputs with their weights and then summing them

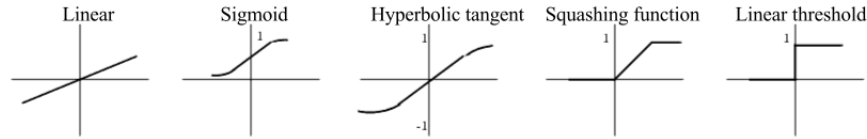


Figure 1.5 – Most common activation functions in MLP neural networks.

making one net input for this neuron (I_j). This one input goes through an activation function (Equation 1.2) which is also called output function. Some of the most common function are shown in Figure 1.5. The activation function output is the neuron output (O_j).

$$I_j = \sum_{i=0}^n x_i w_{ij} \quad (1.1)$$

$$o_j = f(I_j) \quad (1.2)$$

An artificial neuron alone cannot do much, but the calculation power comes when it is interconnected with many other neurons. The connection between any two neurons is associated with a weight (w).

FEED-FORWARD NEURAL NETS (FF-ANN): The way where neurons are interconnected and the data flow inside the network exerts a great influence on the network properties. Inside a feed-forward network, neurons are arranged into layers, where the first layer is called the input layer. The number of units (neurons) inside this layer is always equal to the input of the network. The last layer is obviously called output layer, where the network output limits the number of its neurons. When there are no other layers between input and output layers the network can be called a simple perceptron, otherwise it is called Multi-Layer Perceptron (MLP). Any layer between those two fundamental layers is called hidden layer - there is no obligation on the number of neurons used inside of it. The number of layers in a feed-forward neural network is counted by the number of calculation layers which eliminates the input layer. The input layer does not do any calculations - it just pushes forward the data through the network.

The network architecture meant to specify the number of nodes used inside a hidden layer, showing how many hidden layer are required but also the connection type

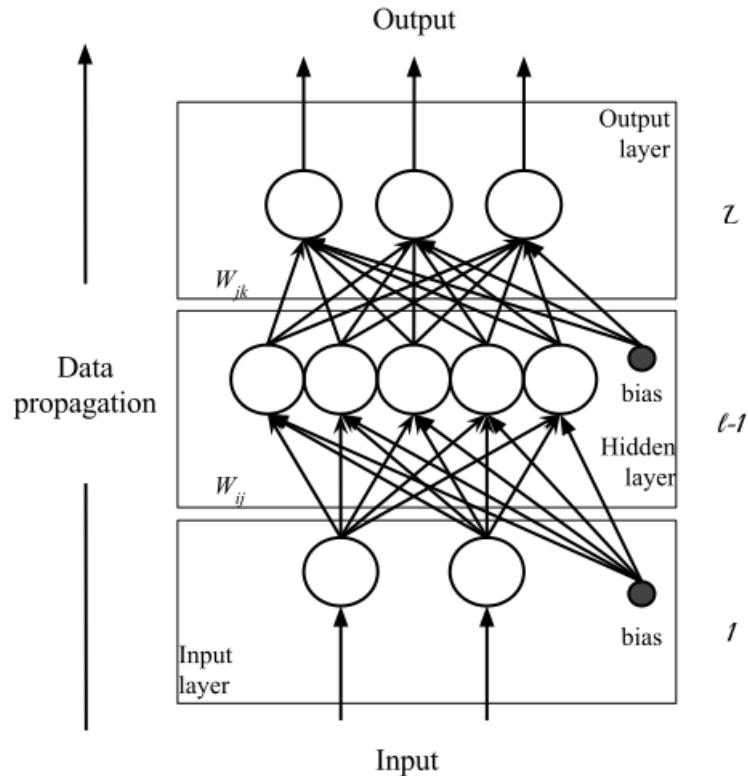


Figure 1.6 – Example of multiple layer feed-forward neural network with data flow direction.

between those layers and nodes. For all feed-forward networks, there is no feedback or interconnection allowed. Every neuron in layer l is only allowed to send information to a neuron inside layer $l + 1$. Figure 1.6 shows the architecture conventions of a feed-forward network. As shown in the figure beside each layer l (where $1 \leq l \leq L$ and L being the number of the output layer) there is a connection from a node outside the layers called bias. In effect, a bias value allows shifting the activation function to the left or right, which may be critical for successful learning.

Equations 1.1 describes the main implementation of feed-forward MLPs. It should be noted that most learning algorithms use the additional class of weights (bias). The biases are values that are added during the calculation of the net input to the activation function. The bias is a constant value and usually equal to 1. As it is a constant value, it increases the capacity of a neural network to solve classification problems allowing the shift/offset of the separation line between classes [17, 50].

As a fact, neural networks cannot be predefined - they have to go through a learning process so they can be used. Depending on the learning algorithm used to

train them, they can learn either very fast or very slow. The performance of the learning process for a given architecture can depend on multiple factors [51]:

- The learning algorithm.
- The data quality used to train the network. The learning data has to be chosen very carefully, it has to include the most information possible with least number of inputs.
- The configuration parameters.

1.2.3 Learning Algorithms

ANN is a powerful calculation tool. This power does not come from programming but from a learning process. Similar to the human brain, ANN needs to be trained and adapted to a specific task. Through the learning phase, the ANN adapts its transfer function to deliver the desired output. Depending on the application, usually one of two major learning types is used to get the expected output from the network - supervised and unsupervised learning [18].

- Unsupervised learning: Those algorithm are used when no specific output is requested of the network, instead the system detects and categorizes persistent features without any feedback from the environment. This type of learning is used for data clustering, feature extraction and similarity detection.
- Supervised learning: Used when the user knows exactly what response has to be associated to each pattern used during the learning phase. During the learning, it is easy to define the error metrics. This output predefinition gives the opportunity to measure the network performance. The difference between the desired output and the calculated one can be used to correct the network behavior in order to decrease this difference (usually called error).

The learning process is usually a repetitive calculation that keeps trying to get a better mapping between the input and the output of the neural network. As the supervised learning usually has a good performance for pattern classification [21], it will get the attention in this subsection. When using the supervised learning, the error between

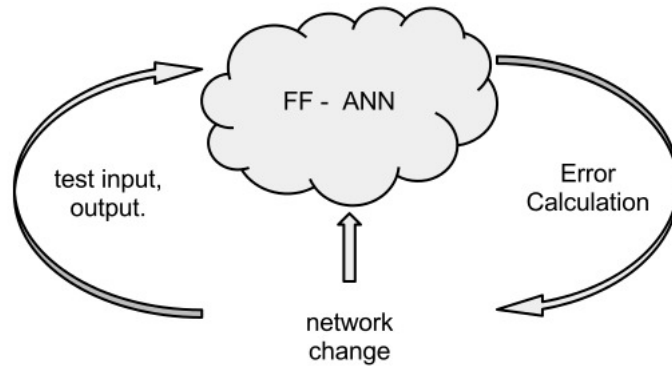


Figure 1.7 – Feedback learning process.

the predefined and the calculated output can be used to modify the network transfer function in order to minimize its value (Figure 1.7).

The back-propagation (BP) Algorithm:

All of the learning algorithms work in order to reduce the networks error. The BP uses the method of gradient descent looking for the minimum error function in weight space. The BP is considered a supervised learning with error correction methodology (corrective learning). As this algorithm is based on the gradient of error function at each iteration step, the activation function must be continuous and differentiable. This is one of the conditions to use back-propagation algorithm. One of the drawbacks of using the error gradient function to calculate the error is being trapped in a local minima and never getting the global minima (Figure 1.8). As the BP algorithm was introduced in the early Eighties, many researchers were engaged to enhance its performance and minimize the disadvantages [18–20, 52]. Due to its wide range of application, the BP is still the most common learning algorithm for feed-forward neural networks [17].

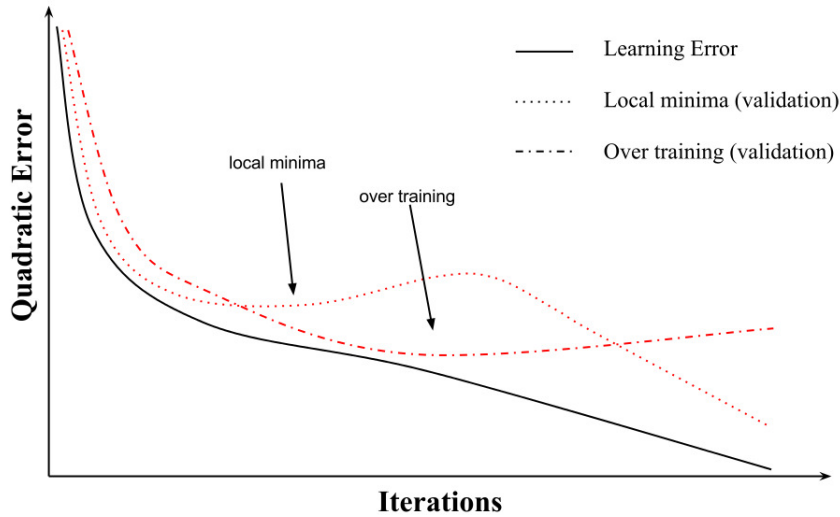


Figure 1.8 – Learning with back-propagation algorithm.

The BP algorithm is based on the Gradient Descent. The Gradient Descent method guarantees that the error will decrease. A repetitive weight adaptation takes place in a manner that will decrease with a condition that Δw (the weight change) is set equal to $-\eta \frac{\partial E(w)}{\partial w}$ and η is small enough ($\eta > 0$). η is called the step-size or **learning rate**. The value of this learning rate can be fix or chosen optimally for each step. In this case it is called *steepest descent method*. The difficulty that faces the Gradient Descent to be applied to multilayer networks is the calculation of the error function ($E(w)$) for each layer other than the output layer. The back-propagation algorithm applied to multiple layer feed-forward network (Algorithm 1.1).

Algorithm 1.1 Back-propagation algorithm steps.

BP algorithm steps for network with only one hidden layer:

1. Random weight initialization for all layer $w_{ij} = rand(a, b)$. Where a, b are two random numbers.

2. Feed-Forward computations:

(a) $I_j = \sum_{i=0}^n w_{ij} x_i$

(b) $o_j = f(I_j)$ where f is the activation function.

(c) $n_k = \sum_{j=0}^m w_{jk} o_j$

(d) $y_k = f(n_k)$

3. Error Calculations:

(a) Partial derivative for output layer:

$$\delta_k = f'(n_k)(y_k - t_k) \text{ where } t_k \text{ is the desired output for neuron } k$$
$$\frac{\partial E}{\partial w_{jk}^2} = \delta_k o_j$$

(b) Partial derivative for hidden layer:

$$\delta_j = f'(o_j) \left(\sum_{k=0}^K w_{jk} \delta_k \right)$$
$$\frac{\partial E}{\partial w_{ij}^1} = \delta_j x_i$$

4. Weight changes:

$$w^{t+1} = w^t + \eta \nabla E(w^t)$$

5. Repeat step number 2 until getting error get an accepted value.

1.2.4 Data Pre-Processing

As it has been often evaluated, during both training and test, the ANN exhibits poor performances when the input data are periodic with zero mean values [50]. The same conclusion can be made for redundant data and very large input vectors as well. Usually ANN weak performances are due to the complexity, redundancy, periodicity or the dimension of input data. To eliminate these drawbacks the input data has to be transformed into a new set of data capable of extracting the maximum amount of information out of it. This data transformation is known as the pre-processing phase. This includes cleaning, normalization, transformation, feature extraction and selection, etc. The output of the data pre-processing is the final training set that enters the neural network (Figure 1.9).

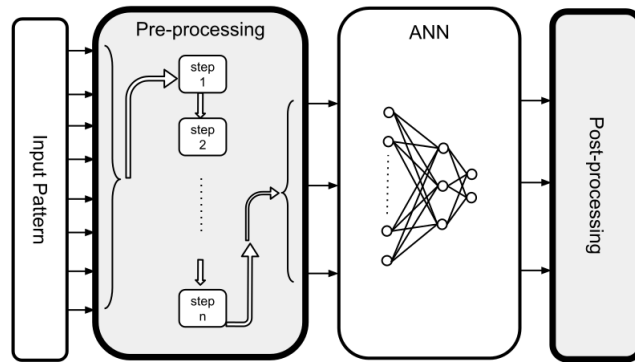


Figure 1.9 – Pre-processing for artificial neural network.

Noise elimination can be one of the most difficult problems while preparing the input data. Usually removing instances with excessive numbers of null feature values can be the first thing to do. The dimension of the input data vector have a significant factor on the convergence time and the problem complexity, which make dimension reduction for large input vector an important step during the preprocessing. Thus some of the most fundamentals of preprocessing steps will be presented [53].

Normalization [24]: This step means to scale down the difference between the maximum and minimum values. This helps to avoid being trapped into the saturation area of an activation function. The most two common methods for this step are:

- min-max normalization: $v = \frac{v' - \min'}{\max' - \min'}(\max - \min) + \min$
- z-score normalization: $v = \frac{v' - \text{mean}}{\text{standev}}$

Where v is the new value, v' old value, min' , max' are the old minimum and maximum value of data, min , max are the new ones, $standev$ is the standard deviation and $mean$ is the mean value.

Missing feature values: This problem is not avoidable, incomplete data is part of the real world data. This information can be missing because it is unknown or lost. Many methods can be found for handling this problem [54]: Method of ignoring instance with unknown feature, concept of the most common feature value, regression or classification methods, etc.

Dimension reduction [50]: When the input dimension vector is very large, it leads to a large number of weight coefficient lists inside the neural network. As a result, a long learning time could occur. Therefore a variety of data compression methods exists. In every compression technique we have to lose data. The user has to make sure to pick up the type that can match his application and minimize data loss. In practice, for classification problems, we often find that beyond a certain point, adding new features could decrease the overall performance of the learning process. One of the important roles of pre-processing in many applications is to reduce the data dimensionality before using it to train a neural network.

1.2.5 Application Domain Overview

The neural networks have a very vast application domain. Over the last few years they have seen an explosion of interest and are being successfully applied in diverse areas as medicine, engineering, geology, biology, etc. They are used in a wide variety of applications where statical methods are traditionally employed. In this thesis, we will focus on the capability of the neural network to classify data. Data classification and recognition is very common in the ANN application, especially when the data relationship is unknown. Electrical machine fault detection, text classification, signature recognition, stock market performance, rainfall prediction and many more are common applications domain for the ANN. The neural network capabilities can be resumed in several points: (i) capability of mapping - they can map input patterns to their corresponding output pattern. (ii) identification of unknown objects. Because they learn by example, they have a capability to generalize. This capability is affected by the training sets and the number of iterations. With more example sets the neural

network knowledge and generalization increase. And (iii) ANNs are robust systems and fault tolerant. They can work with very noisy patterns and even incomplete ones.

ANN for data classification

It is known that most of the ANN architectures and learning algorithms can be used to classify data. The data classification is an operation which puts labels for each input data that has common features. The ANN can perform this task by learning a discriminant function which can separate the different classes. Figure 1.10 shows the example of a two dimensional space problem, where there are only filled and unfilled dots. Figure 1.10b shows one straight line that separates the two types of dots, this can be called a linearly separable space, where the neural discriminant function is linear. In this case, the neural network can learn the discriminant function with only one neuron inside the hidden layer. On the other hand, Figure 1.10a is the non-linear separable spaces, where the discriminant function is more complicated and thus needs more neurons for the hidden layer.

In every field and domain, the input of the neural network must be adapted and prepared in the manner that increases the performance and the ability of the network to learn. An important procedure before using the machine learning in general is the data preparation. This phase usually helps to simplify the discriminant function, decreasing the neurons' number needed in the hidden layer and reducing the learning time.

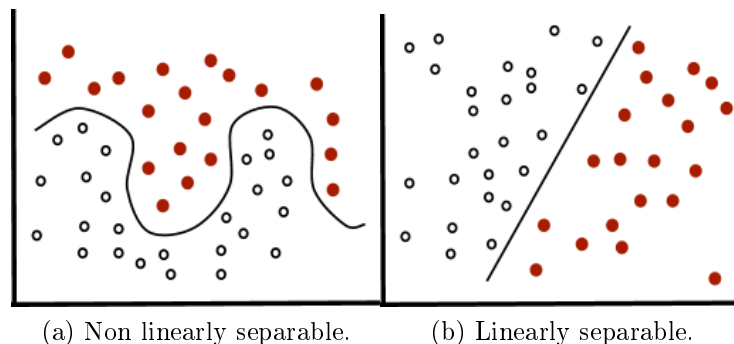


Figure 1.10 – Separability of the different spaces.

Multiple networks

The term multiple artificial neural networks is used for strongly separated architectures [55, 56]. Each of the networks works independently on its own domain. The multiple ANNs are used in a way that divides the input vector into several smaller input vector depending on the type of data. Then each of the small input vectors enters a dedicated ANN. After that, a decision network regroups all of the outputs out of the small networks and make a decision based on a previous training. This make the multiple ANNs works with a large number of neurons in the first place then uses the output of these networks into one final network to generate the final decision.

1.2.6 Summary

This section can be concluded by saying that ANNs have a very large application domain. They can be applied for signature recognition, data mining, data classification, etc. ANNs have a lot of configurable parameters, especially with the feed-forward architecture and the back-propagation learning algorithm. Also all data used as input has to be pre-processed and prepared in order to be able to deliver the maximum amount of information to the ANN. They have been used for electrical machine diagnosis as a supervisor. In this thesis, they will be the only tool for the faults diagnosis and analysis. The feed-forward architecture as well as the back-propagation learning algorithm are among the mostly used architectures and algorithms.

1.3 Discrete Event System Specification Formalism

The Discrete event system specification (DEVS) is a formalism introduced by Zeigler in 1976 [14] and caught the researchers attention since then [57–60]. It describes discrete-event systems in a modular and hierarchical way. A system is considered modular when it has input/output ports permitting interaction with the external environment. The DEVS models are seen as black boxes that send and receive messages from their input/output ports. The formalism distinguishes between the simulation and modeling approaches. The DEVS modeling approach puts in evidence the concepts of modularity and hierarchy by offering two types of models: *atomic models* and *coupled models*. On the other hand, the simulation tree is generated automatically for the DEVS models [61]. The next subsections will describe with more detail the modeling and simulation approaches.

1.3.1 DEVS Modeling

The DEVS formalism defines two groups of models: *atomic models* and *coupled models*. The dynamic behavior(s) of a system is represented with atomic models. The structure of the system is specified using a coupled model. The coupled models are composed of a group of sub-models (atomic and/or coupled) that are interconnected in a manner to deliver the global behavior of a system.

The atomic model

The DEVS formalism defines an atomic model in a mathematical way based on a set of input/output values and ports, two functions define the system behavior while sending/receiving messages and a set of state variables. According to the DEVS formalism "classic with ports", in the atomic models: every port has only one value associated to it. The atomic model (AM) can be defined by the following structure:

$$AM = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, t_a \rangle$$

Where:

- $X = \{(p, v) \mid p \in in_ports, v \in X_p\}$ - the set of inputs ports and values.

- $Y = \{(p, v) | p \in out_ports, v \in Y_p\}$ - the set of output ports and values.
- S is the set of state variables.
- $\delta_{int} : S \rightarrow S$ - the internal transition function.
- $\delta_{ext} : Q \times X \rightarrow S$ - the external transition function where,
 - $Q = \{(s, e) | s \in S, 0 \leq e \leq t_a(s)\}$ - the set of states.
 - e is the time elapsed since the last transition.
- $\lambda : S \rightarrow Y$ - the output function.
- $t_a : S \rightarrow \mathbb{R}_{0,+}^+$ - the lifetime of the state S , $t_a \in [0, \infty[$.

There are two types of events for an atomic model: *external events* and *internal events*. The external events coming from other models, trigger the external transition function and update the system lifetime and state. The internal events correspond to an activation of the internal transition function which updates the model state and lifetime. Directly after that, the output function is executed to generate an output through the output ports.

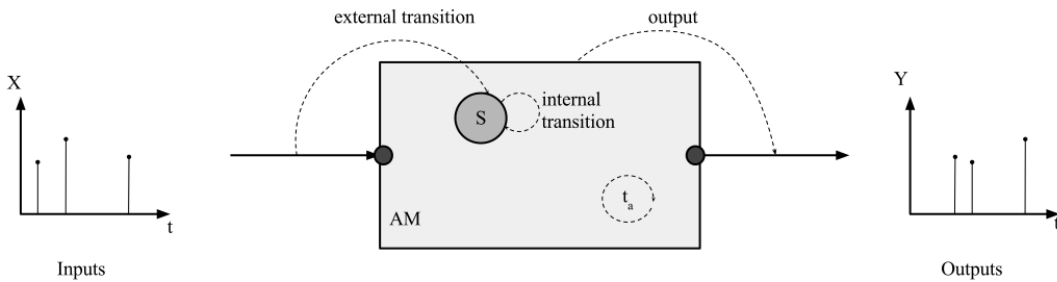


Figure 1.11 – The DEVS atomic model in action.

Figure 1.11 shows the atomic model in action. At any times, the atomic model has a current state s . In the case of no external event occurs, the atomic model will stay in the same state s until the end of the state lifetime. Once the lifetime time elapses ($e = t_a(s)$) the output function as well as the internal transition functions are activated and a new lifetime is calculated for the updated state s . In another scenario, when an external event occurs before the end of the model lifetime (when $e < t_a(s)$)

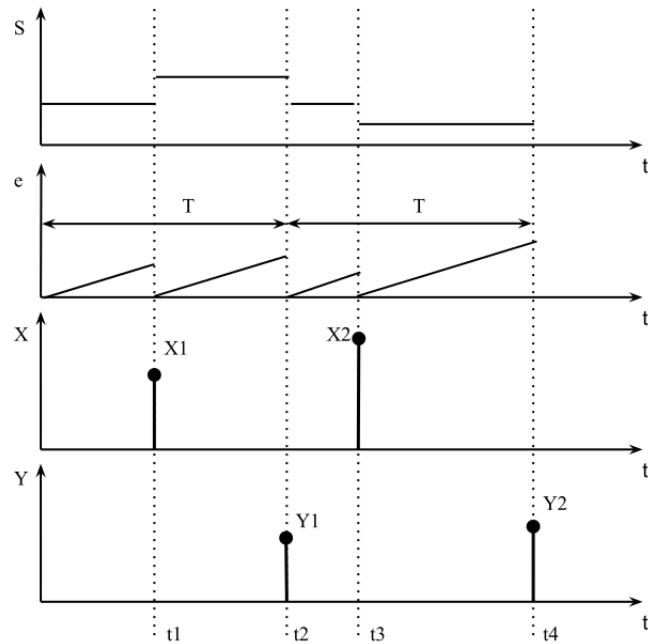


Figure 1.12 – Atomic model state trajectory.

the external transition function is activated. This transition function updates the system state from state s to s' and recalculates a dedicated new lifetime ($t_a(s')$).

If the system lifetime is equal to infinity, then the system stays in the same state s forever without any changes until it receives an external event (in this case the system is passive). On the other hand, if the lifetime is equal to zero, the model will be activated immediately and will execute the internal transition function.

The previous behavior description of the DEVS formalism can be explained in detail with Figure 1.12. In this figure, the external events occur at times $t1$ and $t3$ representing the input trajectory X as X_1 and X_2 . On the other hand, internal events take place at times $t2$ and $t4$. The system state is shown by the trajectory s . It is clear that for every internal or external event the system state changes. The variable e shown as a saw-tooth trajectory represents the time flow by a counter reset to zero each time an event occurs. The output trajectory Y is the result of the output function execution just after the internal transition function.

The coupled model

The coupled model is the proof of the DEVS hierarchical concept. The coupled model consists of a set of sub-models and their couplings. The sub-models can be

either atomic or coupled models. The coupled model's behavior is defined by the behavior of its atomic components and the relation between them. Coupling between these sub-models can be one of three relations (Figure 1.13):

- *External input coupling* (EIC): for coupling between the input ports of the coupled model and the input of its sub-models.
- *External output coupling* (EOC): for coupling between the output ports of the sub-model and the output ports of the coupled model.
- *Internal coupling* (IC): for coupling between outputs and inputs of the sub-models.

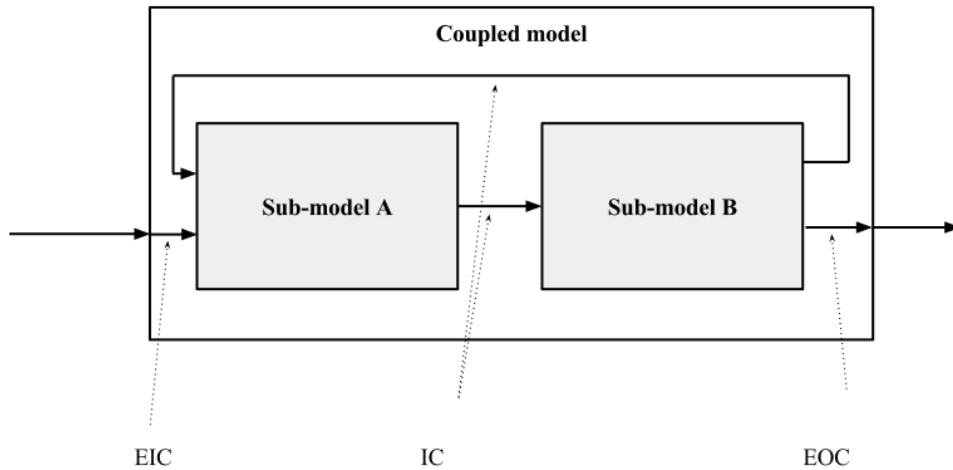


Figure 1.13 – Couplings in coupled models.

Figure 1.13 shows an example of a coupled model composed of two components. These two components can be either coupled or atomic models. This model has one input and one output port connected with the external input coupling and the external output coupling respectively. Any connection between sub-models A and B (Figure 1.13), are considered as internal coupling. Any coupled model can be defined by this structure:

$$CM = \langle X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

Where:

- X the set of inputs,
- Y the set of outputs,
- D the set of sub-components references (names),
- $\{M_i\}$ the set sub-models where for each $i \in D$, M_i can be either an atomic or a coupled model,
- $\{I_i\}$ the set of component influences of i for all $i \in D \cup \{M_{CM}\}$,
- $Z_{i,j}$ the i-to-j output translation function, Where:
 - $Z_{self,j} : X_{self} \rightarrow X_j$ the external input coupling function (EIC),
 - $Z_{i,self} : Y_j \rightarrow Y_{self}$ the external output coupling function (EOC),
 - $Z_{i,j} : Y_i \rightarrow X_j$ the internal coupling function (IC),
- $select : 2^D \rightarrow D$,
 - $select$ chooses a unique component from any non-empty subset E of D :
 - * $select(E) \in E$. The subset E corresponds to the set of all components having a state of transition simultaneously.

With both detailed descriptions of the atomic and the coupled models, the simulation algorithm benefits of the hierarchical DEVS structure to establish an automatic simulation.

1.3.2 DEVS Simulation

One of the most important DEVS properties is that it can automatically generate a dedicated simulator for each model. DEVS distinguishes between the modeling and the simulation in the manner that any DEVS model can be simulated with one generic simulator kernel. Each atomic model is associated with a *simulator* in charge of managing the model's behavior. In the same manner, every coupled model is associated with a *coordinator* in charge of the sub-model's time synchronization. The

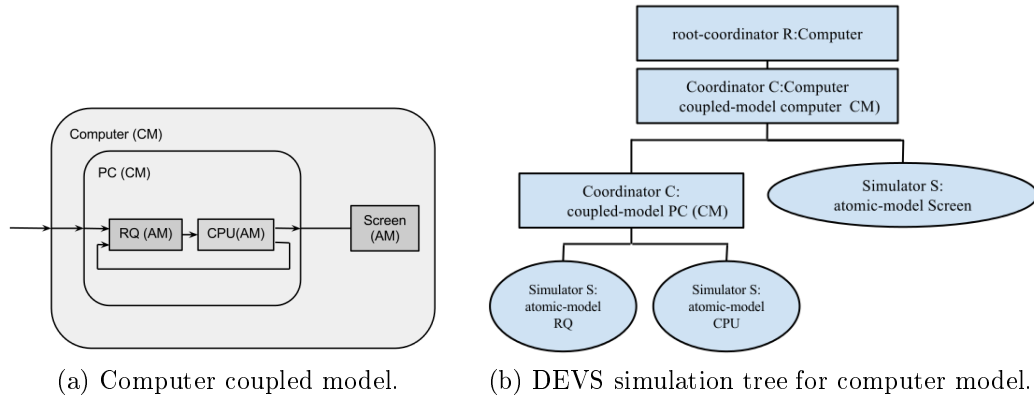


Figure 1.14 – Example of modeling and simulation of DEVS models.

global system can be directed by a special coordinator called the *Root*. Figure 1.14a shows a simple example of a personal computer modeling approach that will be used to explain the DEVS hierarchy (Figure 1.14b).

The «computer» is the main coupled model that contains all other components. As this is a coupled model, it will be associated with a *coordinator*. At the same time as it is the main coupled model the *root-coordinator* will be in control of it as shown in Figure 1.14b. It is clear that for each atomic model there is an associated *simulator* as shown for «CPU» (computer processor), «RQ» (ready queue for instruction execution) and the «Screen model». The coupled model PC is associated to a *coordinator* too.

1.3.3 DEVSImPy Environment

DEVSImPy [15] is an Open Source project (under GPL V.3 license) supported by the SPE (Science Pour l'Environnement) team of the University of Corsica Pasquale Paoli. This aim is to provide a GUI for the modeling and simulation of PyDEVS [62] models. PyDEVS is an Application Programming Interface (API) allowing the implementation of the DEVS formalism in Python language. Python is known as an interpreted, very high-level, object-oriented programming language widely used to quickly implement algorithms without focusing on code debugging [63]. The DEVSImPy environment has been developed in Python with the wxPython [64] graphical library without strong dependences other than the Scipy [65] and the Numpy [66] scientific python libraries. The basic idea behind DEVSImPy is to wrap the PyDEVS API with a GUI allowing for significant simplification of handling PyDEVS models (like the coupling between

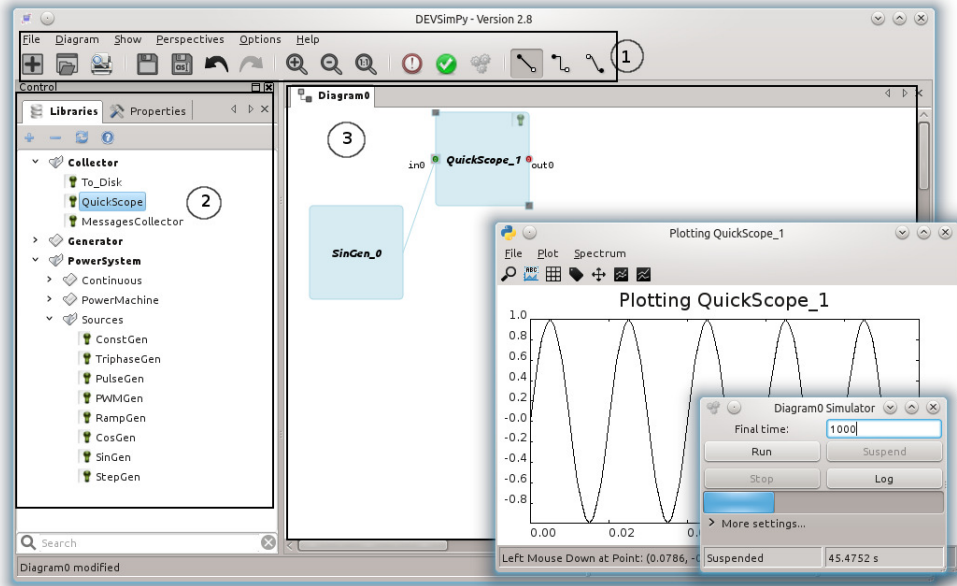
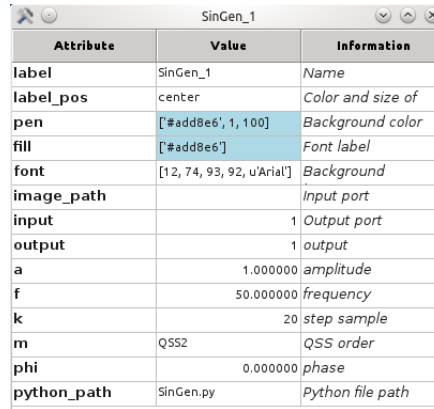


Figure 1.15 – DEVSimPy general interface.

models or their storage).

Figure 1.15 shows the interface of the DEVSimPy environment into three different zones. Zone 1 represents a tool-bar icons and menus that helps the users to create, edit, save, handle and simulate the models diagram shown in zone 2. The models diagram is a graphical representation of the DEVS models and the interconnection between them. The models shown in the simulation diagram in figure 1.15 are an example of a sinusoidal signal generator (*SinGen_0*) and an oscilloscope (*QuickScope_1*) to visualize the output of the *SinGen_0*. This diagram is considered as coupled DEVS model that contains all the DEVS design created by the user. DEVSimPy is based on the use of dynamic libraries composed of atomic (files extension .amd) or coupled DEVS models (file extension .cmd). A list of the different libraries is shown in zone 3 of the DEVSimPy general interface. The libraries can be dragged and dropped from the zone 3 to zone 2 when modeling a system. Every atomic model in DEVSimPy has a dedicated properties panel (Figure 1.16) where some configurations can be made. The properties appear when an atomic model is double clicked. Figure 1.16 shows the *SinGen_0* configuration panel where we can change the output frequency (f), amplitude (a), phase (ϕ), etc. After configuring all models, the user can validate and simulate this modeling by clicking the validation and simulation button in zone 1 of Figure 1.15. DEVSimPy as it is implemented in Python programming language offers



Attribute	Value	Information
label	SinGen_1	Name
label_pos	center	Color and size of
pen	['#add8e6', 1, 100]	Background color
fill	['#add8e6']	Font label
font	[12, 74, 93, 92, u'Arial']	Background
image_path		Input port
input		1 Output port
output		1 output
a	1.000000	amplitude
f	50.000000	frequency
k		20 step sample
m	QSS2	QSS order
phi	0.000000	phase
python_path	SinGen.py	Python file path

Figure 1.16 – Atomic model properties panel.

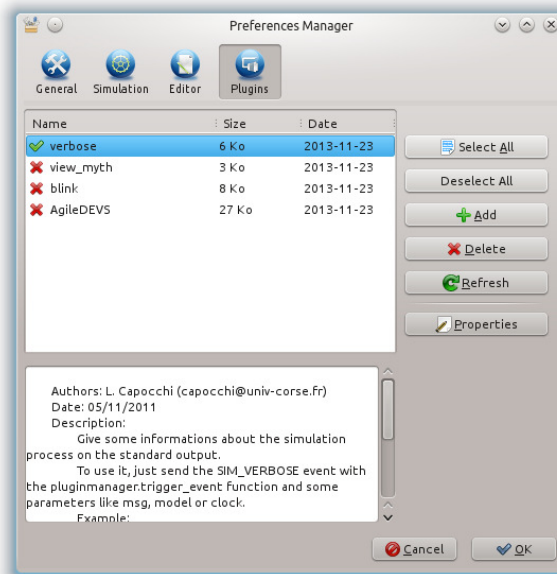


Figure 1.17 – DEVSimPy plug-in manager.

a "simulation suspend" option where the user can modify at runtime some models configuration and resumes the simulation. Figure 1.15 shows the simulator panel where the user can suspend or stop the simulation. There is also a more settings option where the user can choose between different simulator architecture and behavior.

In the computer language, a plug-in is an addition component to an existing software application that adds new features or enables customizations. DEVSimPy has adapted this feature to enable third-party developers to add abilities to the software. The plug-ins use services provided by DEVSimPy to register themselves and exchange data with the plug-in manager. In this way, DEVSimPy is an open ground for developers and researchers to implement new approaches for the DEVS formalism and its

applications using two types of plug-ins: (i) general, (ii) local plug-ins.

The DEVSimPy local plug-in is an addition component to an atomic model that is able to override the basic behavior of the model. The *QuickScope* is an essential atomic models used to display data, where the double click event is overridden in order to show a panel with the curves drawn (Plotting QuickScope_1 in Figure 1.15). In fact the double click shows the properties panel by default, but the *QuickScope* model just overrides that. By the same token a *Dendrogram* model is also implemented in order to deliver a data analysis and show a statistical tree view which will be explained in more details later in Chapter 4.

Also DEVSimPy offers the possibility to implement general plug-ins which are able to change the behavior of all the atomic models present in the DEVSimPy diagram. Any general plug-in can be managed, configured and activated/disactivated from the plug-in manager shown in figure 1.17. A very useful plug-in that can be found by default in the DEVSimPy environment is *Verbose*, where it offer a step by step simulation showing every internal and external transition function for every single DEVS model.

1.3.4 Summary

The DEVS formalism is capable of separating the modeling and simulation approaches of a systems. In that way, each approach work separately. This separation helps the developer to enhance separately each concept without taking care of the other side. The artificial neural network can be implemented using DEVS, getting a modeling flexibility and the ability to be extendable. On the simulation side, DEVS formalism offers a hierarchical simulation tree that can be a good platform to find a new implementation of a comparative and concurrent simulation algorithm. DEVSimPy is an environment that offers a graphical user interface for model creation and simulation. It is a collaborative environment that helps users and developers to share libraries and implemented components and plug-ins. The use of DEVS formalism inside DEVSimPy environment is the key that will help to redesign the neural network and the integration of a comparative and concurrent simulation concept.

1.4 Comparative and Concurrent Simulation (CCS)

1.4.1 CCS Concepts

The Comparative and Concurrent Simulation (CCS) is a concept that allows the concurrent simulation between several experiences [26]. Typically, the concurrent experiments are due to multiple executions of the same program (with different parameters), or execution of different instructions within the same program. This leads us to classify the CCS into two modes: different data paths and different data values. Figure 1.18 shows four execution steps as the main experiment aligned with three

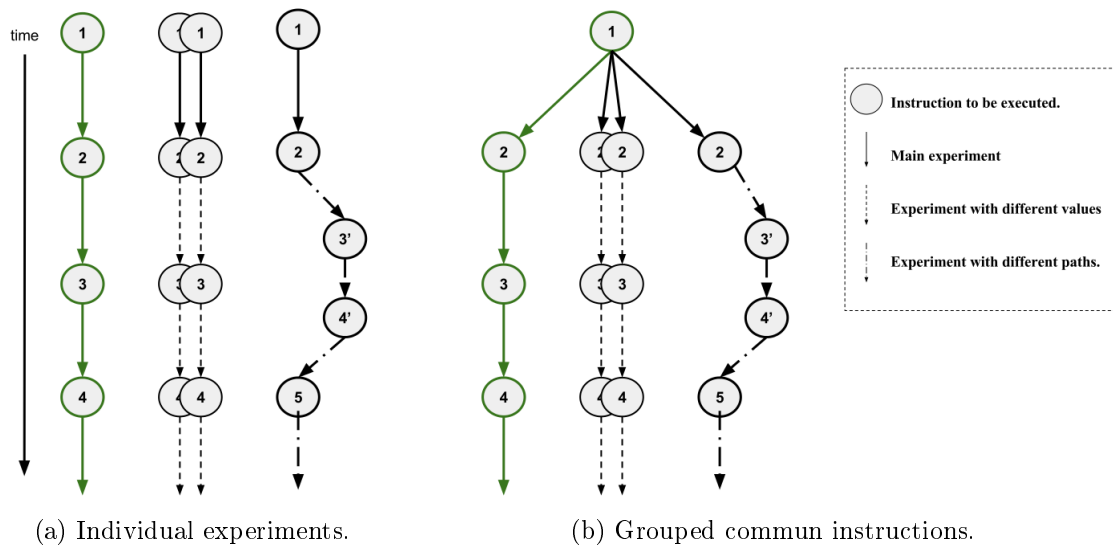


Figure 1.18 – General example of a comparative and concurrent simulations.

other competitive experiments, two with different data values and one with a different execution path.

In figure 1.18a, each experiment executes all the instructions individually. It is a common way to simulate different experiments with different paths or data. The first instruction is redundantly executed with the same values for each experiment. On the other hand, in figure 1.18b, all experiments start from the one and unique instruction simulated only once and for all. One of the advantages of the DEVS-based CCS is the common instruction execution for multiple experiments.

Experiments with the same simulation path are special as they execute the same instructions only with different values. The power of the CCS exists due to several experiences simulated implicitly through one main simulation. The CCS begins with a

reference simulation (R-simulation) that may be the origin of all branched simulations. The branched simulations can be called concurrent simulations (C-simulations). In all manners, any simulation handles data through a path of instructions.

The work presented in this thesis will be more interested in a single path concurrent approach with different data values. The interest is justified by the fact that FF-ANN simulation has only one execution path with different configurations. During the simulation, the data has to follow the same model execution in order to reach the target. One of the benefits of the CCS appears when the user loses the interest to continue simulating an experiment, he can terminate it at any time. To realize this, each experiment has to have a unique key that makes it distinct from any other one. This key is called the experiment-signature or the sub-signature. The signature notation will be widely used in this work in order to accomplish and realize a new DEVS-based CCS approach.

1.4.2 Example on Digital Circuits

Figure 1.19 illustrates the concurrent faults simulation on a circuit with two OR logic gates. Considering the following situation where: the input signal E has the activity $0 \rightarrow 1$ and the signal F has the activity $1 \rightarrow 0$. These activities ensure the evolution of the reference gate A.

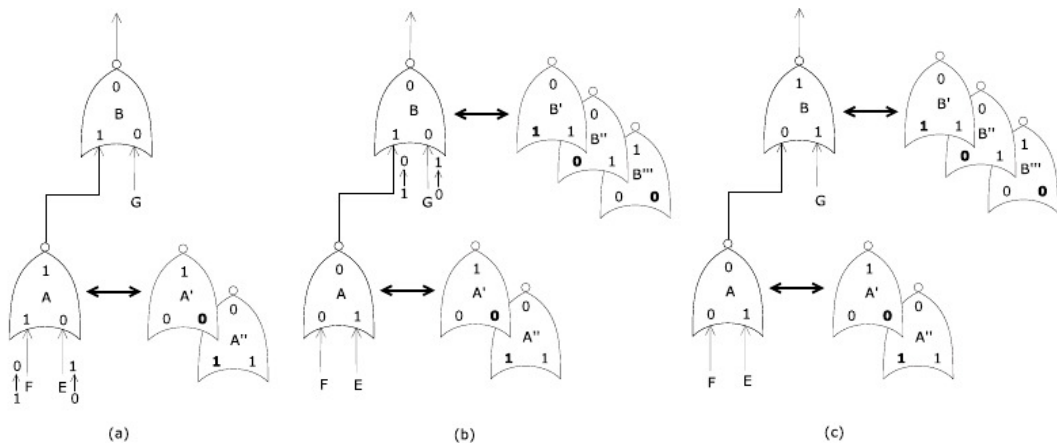


Figure 1.19 – Logic gate fault propagation effect.

The two faults E_0 (stuck in 0) and F_1 (stuck in 1) are the reason for the two faulty gates A' and A'' respectively (Figure 1.19a). The reference simulation starts at the gate A and propagates to gate B. The two faults E_0 and F_1 propagates and involving

the wrong gates B' and B". Also activity $0 \rightarrow 1$ of the input G can be the cause of a new fault called G_0 where the input G is stuck in 0. This fault is the origin of a new faulty gate called B''' (Figure 1.19b). When the simulation ends (Figure 1.19c), as the output values of the gates B' and B" are different from the reference simulation B, the faults E_0 and F_1 are detected. On the other hand, as the output values of the gates B''' and B are the same, the fault G_0 is not detected. This example shows the divergence of the three faulty experiments (A',B'), (A'',B'') and (A,B''') from the reference experiment (A,B).

1.4.3 Simulation Signature

The CCS is an algorithm to allow multiple simulations to be executed at the same time. It is important to note that it is not made for distributed systems or parallel computing. It is an algorithm to enable the ability to compare and execute multiple simulations at a given time step. Figure 1.20 shows a general example of a CCS

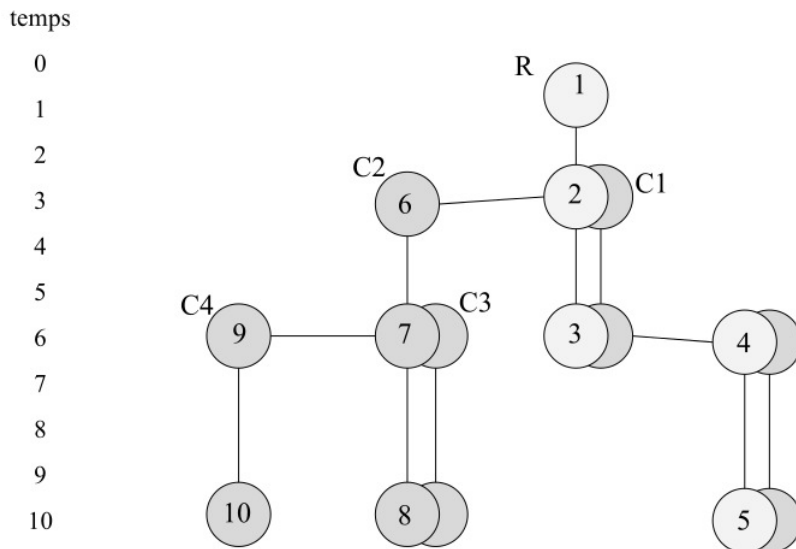


Figure 1.20 – General example of comparative et concurrent simulation.

where the main experiment is represented as R (reference simulation) and concurrent experiments are presented as C_n . An example of a C-simulation that follows the same path as the reference simulation but with different data values is $C1$, while $C2$, $C3$ and $C4$ follow different paths than the original simulation R . At the 6th time step that five different simulations are executed (R,C1,C2,C3,C4). At this time step, all simulations have to be executed and then compared if necessary. Each simulation is

identified by an ID that propagates with the simulation messages or steps. This ID is unique for every simulation, and is a part of a simulation signature. At each step of the simulation, the signature can be changed. It is a very important element that helps to trace simulation changes. A signature can contain elements such as:

- List of executed elements inside a simulation.
- List of changes the elements had during the last executions.
- The time of a new simulation branching (as for the R simulation at step 3).
- The simulations path.

These values are qualified as the **signature** of one experiment. The signature is the data value or configurations that make an experiment unique. It can be accessed with two different permission levels. Classes and instructions that need to read the signature configuration parameters can gain a read-only permission. Read-write permissions are given to classes and instructions that modify data and/or the system configurations. The signature of a simulation can be changed at any step, but it always remains unique and informative. The signature concept is an important key in this thesis, as it will be used for the learning phase of neural networks and compares their performances and their error convergence. Chapter 4 will explain the adoption of this concept to be implemented inside the DEVS formalism and applied and tested for ANNs for electrical machine fault detection.

1.4.4 CCS Properties and Advantages

The CCS is a concurrent algorithm based on the discrete event time domain. It is important to note that the final results of the CCS are proportionally dependent on the number of experiments simulated. The CCS first came to place to remove the serial simulation headaches in the manual work needed by the user to simulate and compare different simulations of a system with different configurations. The fact that the user had to give a lot of attention to the simulations, initial simulations received a lot of attention compared to later ones. At times, this led to neglecting some simulations. The CCS came as a flexible and general software solution for multiple concurrent simulations. Several points can be listed to show the benefits of the CCS:

(i) reducing the simulation time by a significant factor due to the compression of multiple simulations into one general simulation and avoiding manual configuration.

(ii) based on the concurrency nature, the removal or addition of simulation can be done during the execution time. This is allowed due to the unique signature of each simulation. At any time or any step of the execution, each simulation must be able to be identified by its unique ID. This allowed for the concurrency and the comparison to take place.

The CCS can be confused with the Experimental Frame (EF). An EF can be considered as a system that should interact with a model to produce the data of interest under specific conditions [67,68]. The EF only specifies multiple scenarios to the model of interest to produce the desired output. Not to forget some research work are done for distributed and parallel simulations and how can an EF be used [69]. On the other hand the main goal of the CCS is to compare different simulations at certain time points and eliminate any unnecessary execution. One of the big advantages of the CCS over the EF is that common simulation steps can be grouped together to illuminate any redundant execution. There is also some common points between the CCS and the EF: the simulation signature of the CCS can be relatively similar to the EF system structure, where both influence the models behaviors depending on the parameters.

Never to forget that the CCS is mainly oriented to compare different system behavior and to eliminate or add different simulations depending on the user needs. On the other hand the EF is a system that generates different conditions on a model and see the interactions of the model with the system surrounding it.

1.4.5 DEVS/CCS Related Works

One of the first applications of the CCS is the Concurrent Fault Simulation (CFS). The main obstacle to a wide use of this concept is the high complexity to integrate the concurrent simulation algorithms in a simulation kernel. In [70] the author introduces BFS-DEVS, a general DEVS-based formalism for Behavioral Fault Simulation (BFS). The BFS-DEVS a DEVS extension that integrates the CFS algorithms in its kernel and allowing the modeler to specify the faulty behavior of a system using a new faulty transition function. Figure 1.21a extracted from [26] illustrates the relationship

between the CCS simulation kernel and the applications. The kernel implies the use

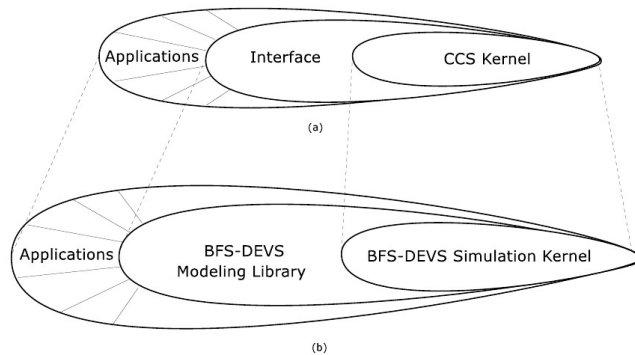


Figure 1.21 – The BFS-DEVS simulation kernel positioning.

of a mandatory modeling interface for a given application. Figure 1.21b shows that an application will be represented by a network of BFS-DEVS components (atomic and/or coupled model) composing the library. The network is directly simulated by the modified DEVS simulation kernel (BFS-DEVS kernel).

BFS-DEVS specifications are very similar with the original DEVS ones. The difference is that any time a faulty event occurs, a new faulty state is calculated by a faulty external transition function (introduced by the BFS-DEVS modeling). If the healthy event occurs, then the new healthy state is calculated by the healthy external transition function. We note that BFS-DEVS models coupling is not changed. But if the fault model contains a structural fault type, this coupling becomes different from the original DEVS coupling. Moreover, we can prove that the property of closure under coupling allowing the hierarchical composition of model is preserved.

The general CFS approach has been applied to digital systems described in the VHDL language using a BFS-DEVS simulator prototype and a library of BFS-DEVS VHDL components. This implementation derives from a simulation kernel conform to Zeigler's specifications. The whole prototype is composed by about 8000 lines of Python code. In order to show the validity the approach, a sub-set of the VHDL ITC'99 benchmarks [71] has been chosen.

1.4.6 Summary

The CCS is a concept that gives the user the option to simulate multiple simulations at the same time with only one processor. It can handle different simulation paths and

values too. This concept, allows a better supervision and analysis for all experiments within a single simulation execution. The CCS is applied in different domains like concurrent fault simulation and was also introduced to the DEVS formalism with a creation of a new simulation kernel (BFS-DEVS simulation kernel). At the mean time, modelers still need a generic concurrent behavior that can be applied without the change of the simulation kernel.

1.5 Conclusion

The diagnosis of electrical machines has been an important research subject for more than a century. Electrical faults are considered the most common ones. In this work, electrical faults, especially inter-turn short circuit in both stator and rotor sides are investigated. In the last years, more research works were focused on AI in order to improve performances of the traditional model-based methods. The artificial neural networks have a very large application domain. The ANNs have a lot of configurable parameters, especially with the feed-forward architecture and the back-propagation learning algorithm. They have been used for electrical machine diagnosis as a system supervisor, a further step would be to use the ANNs as the only tool for the faults diagnosis and analysis. The DEVS formalism is capable of separating the modeling and the simulation approaches. In this way both sides can be modified separately and transparently to each other. DEVSimPy is a DEVS environment that offers a graphical user interface for the modeling and the simulation. It also offers a plug-in manager capable of modifying the simulation or the modeling behavior. The comparative and concurrent simulation is a concept that offers the simulation of multiple experiments implicitly within a single simulation. It has been used to in a concurrent faults simulation of the digital circuits and also integrated with the DEVS formalism creating a new simulation kernel called the BFS-DEVS.

Chapter 2

DEVS-Based ANN

2.1 Introduction

Artificial neural network has been thought to be a black box capable of resolving problems that are hardly written in mathematical forms. Therefore, the configuration of this revolutionary computing remains a hard task to be generalized since it depends on the application complexity. On the other side, DEVS formalism gives the opportunity to redefine a model in a way that allows interaction with the structure and/or the behavior of the model with an automatic simulation. This can be done due to the hierarchical and modular aspect of the DEVS formalism using different model types and an automatic simulation algorithm.

The nature of the feed-forward ANN architecture is based on discrete messaging (events) between neural layers or individual neurons. In this way, modeling ANN in a discrete event formalism is strongly considered. Many DEVS systems integrates the usage of the ANN, but almost none of them redesigned the whole neural architecture inside the formalism [72]. Most often neural network is presented into the DEVS hybrid systems as one atomic model that can interact with different models. The difference in this work is that neural network design will be fragmented into several atomic models inside the DEVS formalism to create an ANN library.

This chapter is organized as follows: (i) mapping between ANN and DEVS and a comparison between three different levels of description, (ii) the modeling of the chosen level inside DEVS formalism, (iii) the simulation process and how the learning process is controlled. Lastly, is the creation of ANN library inside DEVSImPy environment.

2.2 ANN/DEVS Mapping Study

A feed-forward neural network is composed of several layers, inside each layer there are neurons. Neural networks can be seen from three different levels of description. The first shows the ANN as one black box with inputs and outputs. The second as several neural layers connected to each other but the layers are still black boxes. The third view, goes deeper inside the smallest neural unit, as it shows all connections between every neuron in the network. This study will show the advantages/disadvantages for each level of description.

In the last chapter, figure 1.6 is an example of multiple layered feed-forward ANN that shows connections between all neurons and the data propagation from the input to the output of the network. Identifying all the basic elements of the neural network is the first step of the modeling. Another important step is the implementation of the learning algorithms, which will be discussed later in this chapter. The DEVS model design could play a very important role in the ANN flexibility and simulation speed, which makes the modeling very important to get the best performance of the simulation.

The mapping study begins with the identification of the basic elements for a feed-forward artificial neural network calculations:

- **Input data list:** The input of the neural network, is represented with a list of non-calculations neurons. It is recommended to process the data and adapt it to match the recommendations for ANN input (Section 1.2.4).
- **Output data list:** The output of the neural network. It is represented with calculation neurons.
- **Neurons:** Can be considered as the smallest unit of a neural network. Calculations neurons are composed of a **transfer function** and an **activation function**. Non-calculation neurons are just data transporters from one side to another.
- **Neural connections:** Can be considered as the message transporter between two neurons. For each connection there is a coefficient that insures the strength of this connection between two specific neurons.

These four elements are required for calculations, but as previously mentioned, neural networks need to be trained in order to be used. The next subsection will propose different modeling levels approaches for the DEVS-based ANN models.

2.2.1 Neuron Architecture Level

At this level, the basic atomic model of the network will be one neuron. Figure 2.1 shows the modeling design that represent one neuron as an atomic model (AM). There are two different types of neurons. For the input layer (I) all neurons are non-calculation neurons. This type of neurons do not make any calculations (no transfer nor activation function) as it is only responsible for sending input data to the next layer (hidden layer).

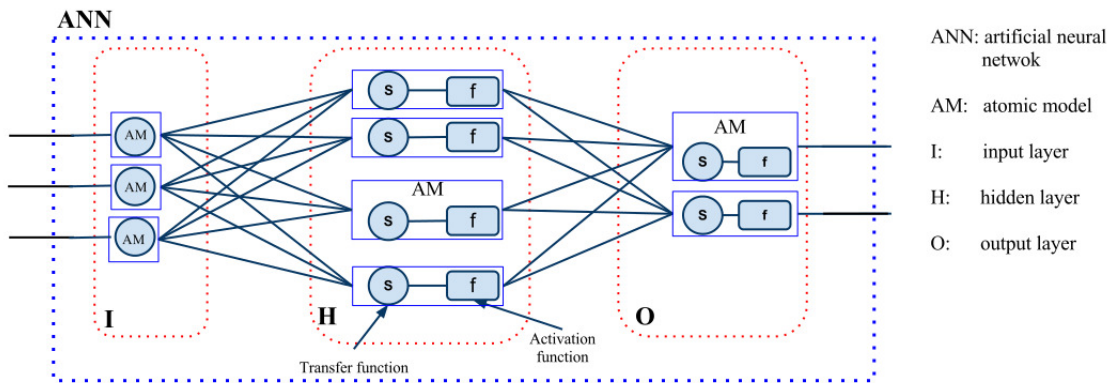


Figure 2.1 – Design of one neuron modeling level.

The second type of neurons is the calculation neurons, which can be found inside the hidden (H) and output (O) layers. Each model has inputs equal to the number of neurons inside the previous layer. After it receives all input messages, it multiplies every input with the corresponding weight and the sum is calculated (transfer function). The activation function is then computed to produce one output, which is the output of this particular neuron. In a larger view, the DEVS atomic model will represent the neuron model presented in the previous chapter in Figure 1.4.

Advantages of neuron architecture level:

- The graphical representation is really the highlight of this approach. Visually (inside a GUI) the user can find the most detailed vision of his constructed

network. Just by looking at the network, the number of neurons inside the hidden layer can be noticed.

- Neuron connections can be manually controlled as each neuron is defined individually, making the connection architecture more flexible.
- In some advanced work, a dynamic DEVS structure [73] can be applied to delete or add neuron models in order to change the ANN configuration within the simulation run-time.

Disadvantages of neuron architecture level:

Unfortunately, there are some disadvantages to this modeling approach:

- Messaging between the models could be an issue. As each neuron model sends an individual message to all neurons in the next layer, this can produce a very large number of messages which can easily slow down the simulation. Additionally, this design produces a very large number of messages, it sends redundant messages as each neuron sends the exact same message to all next layer neurons.
- Parameters configuration is one of the important steps for using an artificial neural network. Configuring the network for the learning process for this design requires accessing each individual neural atomic model and changing the configurations one by one for each layer. This might not be very practical for any user as the configuration needs to be as easy and as fast as possible.

2.2.2 Layer Architecture Level

Going up one level from the neuron architecture leads to the grouping of all neurons inside one DEVS atomic model. Because two types of layers exist - computational/non-computational layers - two different atomic models have to be implemented (Figure 2.2). The first model (A1) has the same role as the non-calculation neuron models in the previous approach, as it is responsible for forwarding the data without exercising any calculations. The second model (A2) will inherit the functionality of the neural calculation model (present in hidden and output layers) in order to calculate the transfer and the activation functions. That leaves us with two types of DEVS atomic

models representing two different layer types (calculation and non-calculation layers). The Input layer is considered as the non-calculation layer (A1) and both, the hidden layer(s) and the output layer (A2) are considered as the calculation ones.

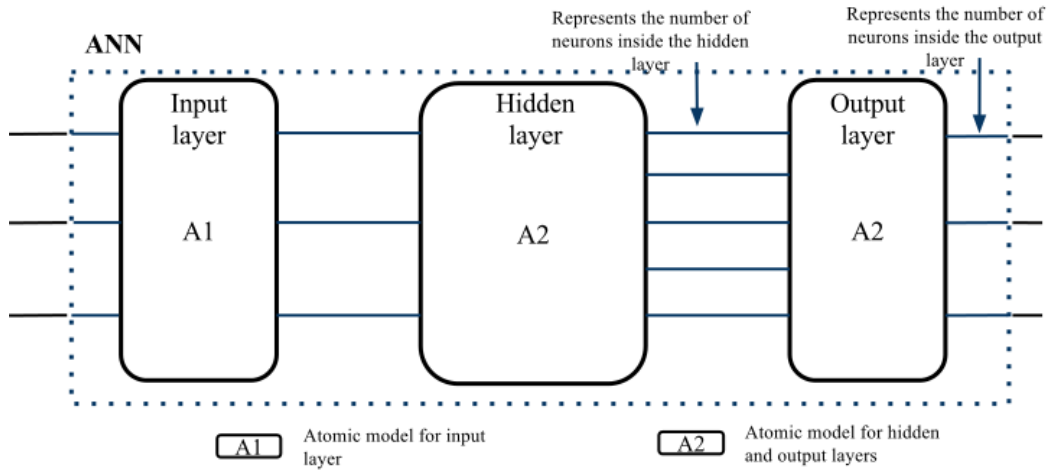


Figure 2.2 – One atomic model per layer modeling approach.

This modeling approach will introduce several changes compared to the previous one. First visually, as shown in Figure 2.2 the connection between layers is equal to the number of neurons inside the layer that sends outputs. In other words, if the hidden layer contains five neurons, the connections between the hidden and the output layers will be equal to five.

Advantages of layer architecture level:

- Graphical representation still exists even after encapsulating the neurons models inside one model. The number of neurons inside any layer can be shown by the number of outputs of the same layer.
- Configuration is much easier compared to the previous approach. Basically, each layer has a different configuration than the others, but neurons inside the same layer are configured with the same parameters, which fits perfectly with this modeling approach.
- Simulation time and CPU usage are reduced over the previous neuron approach. This is due to the decrease in number of atomic modes simulated.

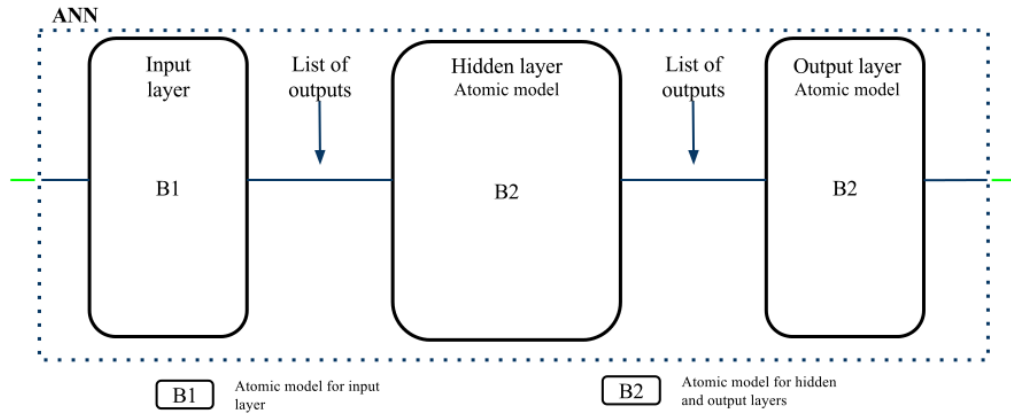


Figure 2.3 – Reduced messaging approach.

Disadvantages of layer architecture level:

- At this level, the ANN has to be a fully connected network. That means, any message received by a calculation layer will be transmitted to all neurons in the layer.
- The large network with a lot of neurons will still need a lot of CPU usage as each neuron sends at least one message.

2.2.3 Layer Level with Reduced Number of Messages

Inspired by the improvement that the layer architecture approach made in front of the neuron level, a new approach can be considered. This approach is the same as the layer approach, but with a reduced number of messages. This might increase the performance of the simulator and reduce the amount of messages sent between models.

Figure 2.3 shows that between neural layers, only one message is sent. This message contains the list of inputs necessary for neural calculations. By this message reduction, each layer sends only one message that contains all outputs. In this case, when a layer receives a message, all neurons inside this layer get all inputs as this confirms a fully connected feed-forward ANN.

Advantages:

- Faster simulation from the two previous models because the number of message is reduced to be one, with a list of the output values.
- Configuration has the same advantages from the layer architecture level.

Disadvantages:

- In the case of implementing the approach with a graphical user interface, there will be no indication of the number of neurons inside any layer.
- As any layer sends only one message with a list of values, it will be even harder than the previous approach to change the neuron interconnection as it will remain a fully connected network. If it is not the case, the user has to change the internal code of the layer atomic model.

2.2.4 Comparison and Selection

To be able to choose between the presented modeling approaches, some important point has to be clarified. First, the amount of calculation inside either the Hidden or the Output layer is fixed by the number of neurons inside the layer despite the number of messages received. The transfer function can't be executed until all messages are received, which makes the simulation's calculations only can be changed when the number of received messages changes. With the neuron architecture level, each neuron sends individual messages to all neuron in the successive layer. On the other hand each neuron receives individual message from each of the previous layer. The layer architecture level, regroups all of the individual messages into only one list to avoid redundant messages. As shown in Figure 2.4, the individual neuron sends the same data through multiple message, on the contrary the layer architecture sends the value once and for all. The third proposed architecture regroups again the multiple values and sends them in a single message in a form of a list which eliminates completely the visual advantage offered by the other architectures.

From the previously described advantages and disadvantages for each of the proposed architectures, the architecture that fits ours needs is the second architecture

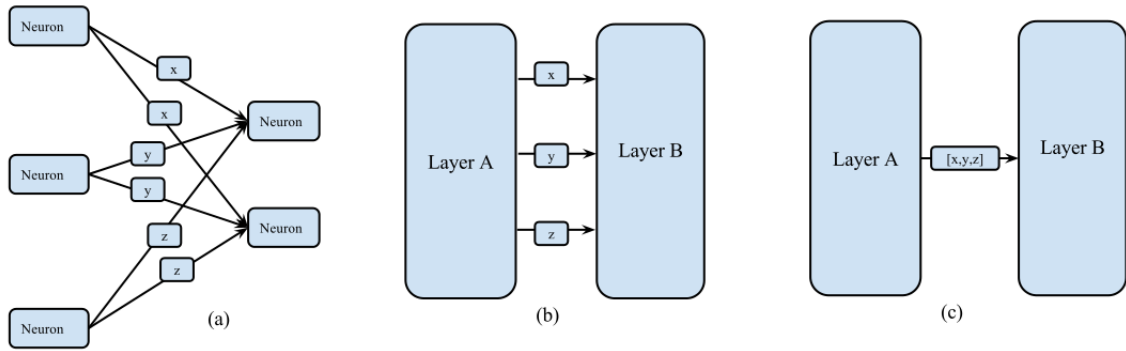


Figure 2.4 – Messaging comparison between the three proposed approaches.

(layer level). The choice compromises between the performance and the visual advantages. From the visual the user can detect the number of neurons inside the layer and their is no redundant messages.

2.2.5 Summary

The choice of modeling can really affect the simulation performance, flexibility and extendibility. One of the visions adopted in this thesis is the extendibility. By extendable we mean to be able to introduce design enhancement for the simulation, and reduce the work redundancy to import a better algorithm or design. In this manner, the choice of artificial neural network modeling with DEVS formalism will rely on a design that ensures: a visible architecture of the neural network, modularity and capability of integrating multiple learning and optimizations algorithms.

After comparing the three proposed modeling approaches, the layer architecture is our choice due to its low CPU usage and the good graphical representation of the network. The next section will describe in more detail how each model will be implemented, as well as the timing of the learning phase.

2.3 ANN/DEVS Modeling

The main idea of this section is to show the artificial feed-forward neural network with an example of back-propagation learning algorithm (Subsection 1.2.2) in a new DEVS-based model. The first step is the introduction of the first (*Input*) model as the non-calculation neural layer. This layer is in charge of forwarding the input to the network. The implementation of the calculation layers (hidden and output) is the second step. The last step will be the implementation of the learning algorithm. In the same manner as the feed-forward calculation, the learning algorithms should have the same modularity and flexibility. For the purpose of modularity, the learning algorithm will be implemented separately from the network calculation. As a result, there will be models for feed-forward calculations and others for learning.

2.3.1 DEVS-Based ANN Design

The artificial neural network is composed of two types of layers - calculation layers and non-calculation layers. The non-calculation layer represents the *Input* layer in a ANN and the calculation layer represents hidden and output layers. The number of hidden layers can vary from one to several layers. Generally, one hidden layer is enough to solve most non-linear problems. In this thesis, the DEVS-based ANN will be mapped into four different atomic models that will assure calculations and learning.

Figure 2.5 shows the proposed DEVS-based neural network modeling design. Focusing on the design blocks, four blocks can be found representing the ANN layers and separate training (learning) models. The *Input* layer is considered as the non-calculation layer. Hidden and Output layers are the calculation layers. *Error-Generator* and *Delta-Weight* are the two models that represent the ANN learning phase.

As shown in Figure 2.5, a training layer can be added or removed depending on need. The advantage gained from of this design is that the training layer is pulled out to be separated of the three known layers. This separation means that the learning algorithm can be replaced by changing the layer models. On the other side, with already learned networks, the training layer can be completely removed.

To notice that in figure 2.5 all ports with a circle are ports used only during the

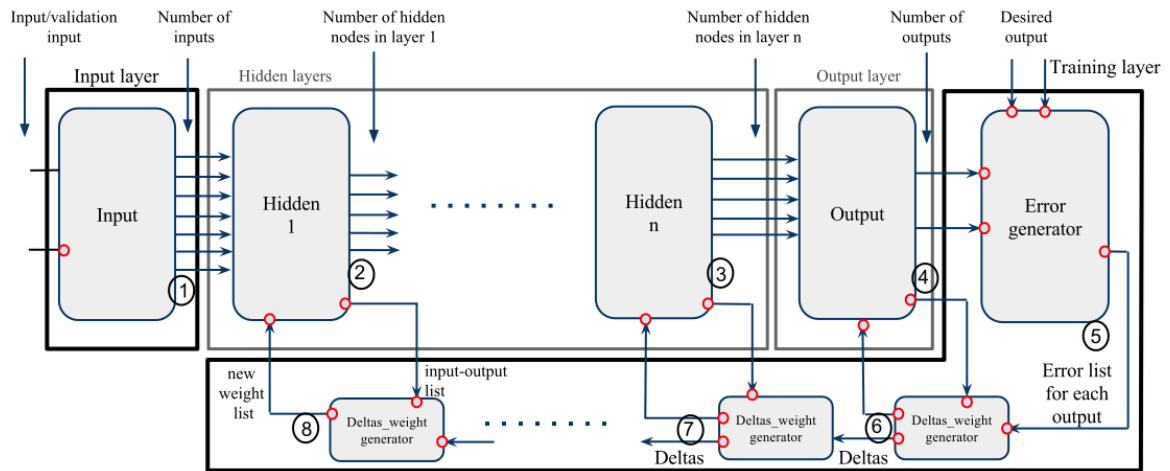


Figure 2.5 – DEVS-based neural network modeling with the feed-forward architecture and the additional training layer.

training phase. As shown in the Figure 2.5, the connections between the training layer and the calculation models are for learning purposes only. In that manner, when the training layer does not exist, the calculation of the ANN will continue to function normally and provides output without any error feedback.

As already known, feed-forward network with back-propagation learning algorithm has to feed-forward all calculations and then back-propagate the error for learning and weight changes. In the presented modeling (see Figure 2.5), the *Input* atomic model starts to push data to the hidden layer (shown as step 1 in the figure), then each Hidden layer (including the Output layer) calculates the activation and transfer functions. Each hidden\output layer then sends data to the next layer and sends necessary learning data to the training layer (steps 2,3,4). The *Error-Generator* model starts the back-propagation calculations by sending the error (difference between desired and real output) to the *Delta-Weight* models (step 5). The *Delta-Weight* models recalculate the weight-list from the error values with learning algorithm. After that, the *Delta-Weight* model continue to back-propagate the error and update its associated model (steps 6,7,8). The entire cycle explained above is considered as one learning iteration that is repeated (steps from 1 to 8) as many times as the training process needs.

2.3.2 Feed-Forward Calculations Model Set

Feed-forward calculations are the result of the three known layers (*Input*, *Hidden*, and *Output*). Those three types of layers will be implemented in two classes of DEVS models - non-calculation model for the *Input*, and calculation models for hidden and output layers.

Non-calculation layer atomic model

The DEVS-based neural network first layer model is responsible for receiving inputs patterns and pushing them forwardly to the network. In this manner, it controls the data flow inside the network. Bearing this responsibility, the *Input* layer is the simulation manager.

There are two main goals for this model. The first is to receive all patterns that will enter the neural network for all purposes (learning, testing or usage). The second is to push the patterns sequentially to the hidden layers (pattern generator).

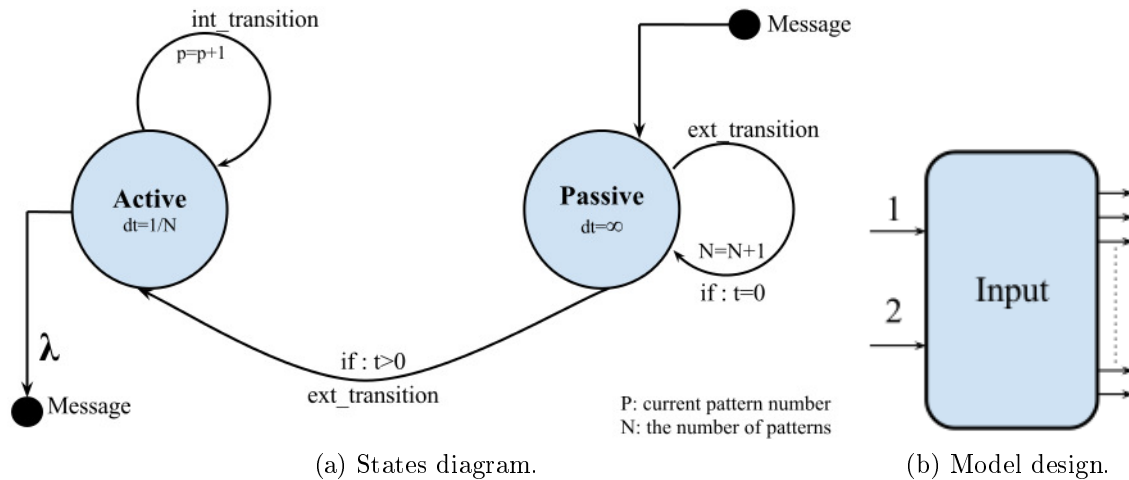


Figure 2.6 – The non-calculation (*Input*) layer DEVS atomic model.

The non-calculation layer atomic model has two input ports (Figure 2.6b) to receive two sets of input. In a learning scenario, the first port receives the training patterns list and the second port receives the validation patterns list. In a normal scenario, only the first port is used to receive the patterns list. On the other side, this atomic model can have as many outputs as can be found in the received patterns.

As a DEVS atomic model, the non-calculation (*Input*) layer model has different states that it can transit between during the internal or external transition functions.

By default, this atomic model is in a passive state with an infinity state life-time (Figure 2.6a). When a message is received during the simulation ($t > 0$) the external transition function changes the model state to an active state and it remains active with a state life time equal to $1/N$, where N is the number of patterns used for learning.

Figure 2.7 describes the sequence that the *Input* layer will follow during the simulation. At the beginning, the *Input* layer is a passive model, (I) and will wait to receive patterns necessary for the simulation. Once it receives the first event, (II) the external transition function model transfers the model from a passive state to an active state. The time advance function determines, (III) based on the number of patterns, when the next internal transition will take place. The model remains active (as a generator) and sends each dt an output until the simulation time decided by the user ends (IV). For the learning process, the simulation time represents the number of iterations decided by the user.

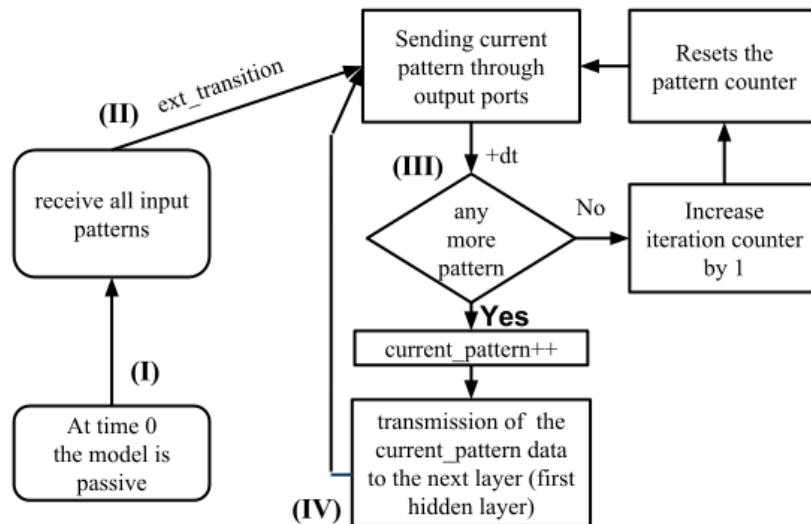


Figure 2.7 – Block diagram for *Input* layer DEVS atomic model (Non-calculation layer model).

To conclude, the *Input* model will have two input ports (Figure 2.6b). The first is to receive patterns for a normal or a learning process. The second is to receive validation patterns that are calculated during the learning process. The patterns are composed of lists, and each list contains the input for the neural network. The number of outputs ports is equal to the number of values inside each input pattern. Finally, the *Input* model is the manager of the simulation.

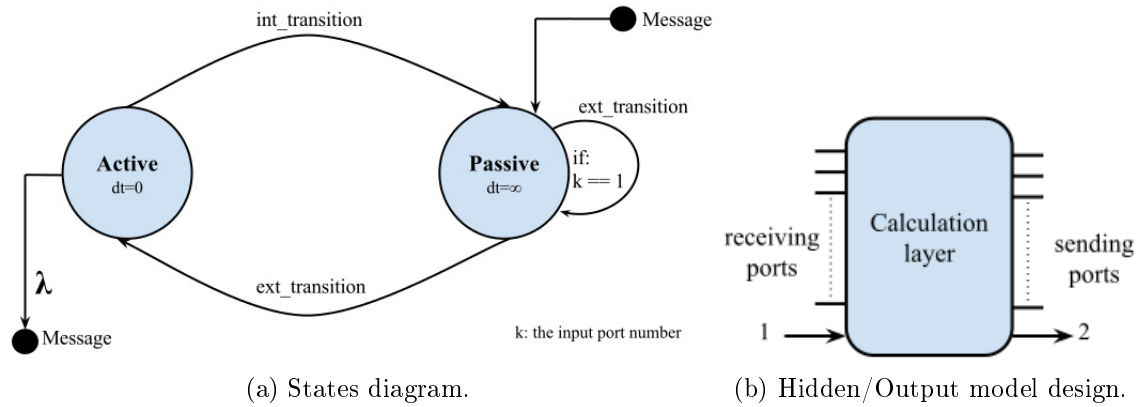


Figure 2.8 – Calculation layer DEVS atomic model.

Calculation layer atomic model

The calculation layer model represents any hidden or output layer, as both layers have the same behavior. This model remains passive at all times due to the single responsibility to calculate the activation and the transfer functions after receiving input messages. A calculation layer contains two main functions: the activation and the transfer functions.

Figure 2.8b shows the calculation layer with multiple input and output ports with two additional special ports at the end of each sending and receiving sides. The number of input/output ports depends on the number of incoming messages and the number of neurons inside the layer respectively. The last two ports on both sides (sending/receiving) will be reserved for learning purposes. During the learning phase, the model receives an updated weight list from the last port on the receiving side (port 1). By the same token the last port (port 2) sends all necessary information needed to perform the learning algorithm. The needed information is in steps 3 and 4 in the back-propagation algorithm (1.1) and can be summarized with the inputs/outputs of the layer and the weight list. All information needed for learning will be stored in the learning model set.

As an atomic model, the calculation layer will oscillate between two different states: active and passive states as shown in Figure 2.8a. The initial state is passive and the model always waits for new data to calculate the activation/transfer function. The model's state is transferred from passive to active when the external transition function receives a message from any port other than port one, and the feed-forward calculation continues (Algorithm 1.1 step 2). If the model receives a message from

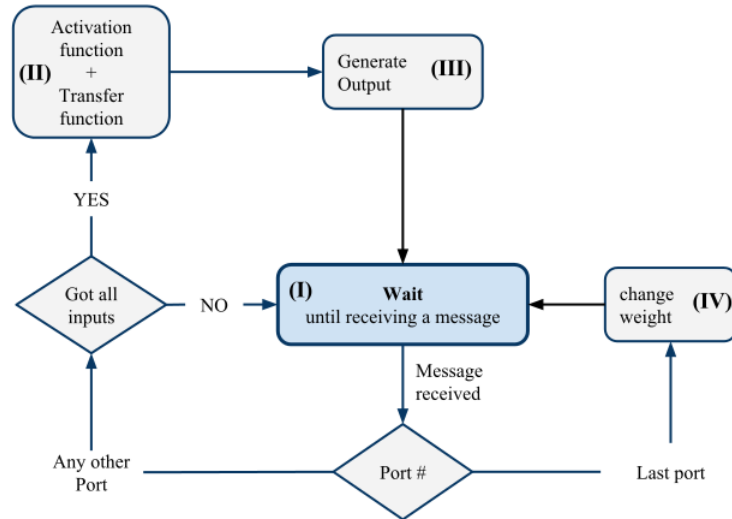


Figure 2.9 – Block diagram for calculation layer DEVS atomic model.

port one, that means it receives a weight list update (learning phase) and it remains in the passive state.

The block diagram of this atomic model is presented in Figure 2.9. The calculation layer model is a passive (step I) and only becomes active when it receives a message. When the atomic model receives a message from any port other than port number one, it verifies if all messages have been received. If yes, the activation function is calculated (step II) followed by the transfer function for all neurons inside this layer. After that, all generated output will be sent through the output ports (step III). In another scenario, when a message is received on port one, the weight list will be changed (step IV).

The four steps explained above represent the entire calculations layer, which are all the hidden and output layers.

2.3.3 Back-Propagation Learning Model Set

The back-propagation algorithm is a learning algorithm commonly used for feed-forward neural networks. A description of the algorithm is presented in section 1.1. In this subsection, the algorithm will be split into two basic models. The first is the *Error-Generator* model and the second is *Delta-Weight* model. The two atomic models will be responsible for the learning phase of the feed-forward calculations. As a result, the modeling approach presented in this thesis splits the learning models to ensure the separation between learning algorithms and neural calculations. Any

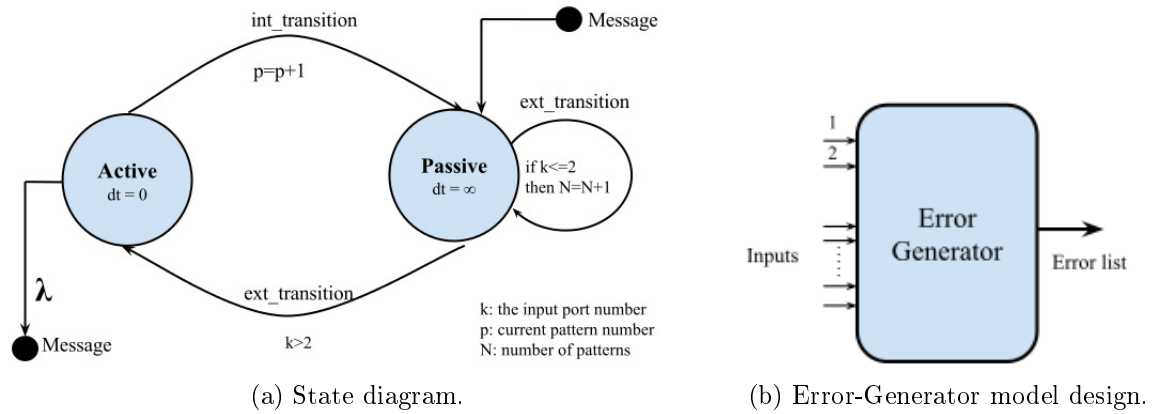


Figure 2.10 – Error-Generator DEVS atomic model

changes in the learning algorithm will not affect the calculation models and vice versa.

Error-Generator atomic model

As noticed from above, all models other than the *Input* layer are passive - meaning they are activated only when a message is received. The *Error-Generator* will make no exception, as it is a passive model. This model is the first to be executed among the learning model set. It receives the real output of the neural network from the output layer atomic model, and then it compares the real output to the desired output that the ANN should have. In an ideal situation, the error between the real and the desired output is zero. Unfortunately, there is always a percentage of error for each of the outputs. The error list is sent to *Delta-Weight* atomic models that re-calibrate the network for a better performance.

Figure 2.10b shows the atomic model input/output ports in detail. In fact, ports one and two are the equivalent of the input ports of the *Input* layer model (see Figure 2.6b). The first port receives the desired output list for the learning patterns. Similarly, the second port receives the messages containing the desired output for the validation patterns. With the same token as the *Input* model, the *Error-Generator* model is initialized with patterns received from ports one and two. However, it remains passive even after receiving all necessary patterns needed for the learning process. Furthermore, the *Error-Generator* becomes active only when it receives messages from any other port then the first two ports (Figure 2.10a). Only when it receives all necessary inputs, equation 2.1 is calculated to generate the error E_k for every output of the network, where y_k is the real output and t_k is the desired output for each port

k .

$$E_k = t_k - y_k, \quad \forall k \in [0, N] \quad (2.1)$$

$$E_{Total} = \frac{1}{2} \left(\sum_{k=0}^N (E_k)^2 \right) \quad (2.2)$$

Where N is the number of neuron in the output layer.

To visualize the performance of the learning process, the combined error of all outputs has to be tending towards zero. The ANN might not be able to learn with the given configuration if the error diverges from zero. This combined error is calculated with equation 2.2. It is important to note that the individual output error is squared, otherwise positive and negative values may cancel each other out.

When the *Error-Generator* becomes active, it calculates the error between desired and real outputs, sends a message with the error list through the output function (λ) and then through an internal transition function it goes back to the passive state waiting for a new message (Figure 2.10a).

The second scenario is when the model receives messages on ports one or two. In this case, it remains in the passive state, and the external transition function accumulates the input from both ports and increments the patterns counter by one ($N=N+1$) as shown in Figure 2.10a.

In addition, the individual output error produced by the *Error-Generator* in the form of a list is sent directly to a *Delta-Weight* model to continue the learning process.

Delta-Weight atomic model

The *Delta-Weight* model represents the core of the algorithm. As already noticed, the *Error-Generator* only calculates the difference between the desired and the calculated output, called error. The full DEVS-based ANN modeling approach presented in Figure 2.5 shows that every calculation layer model (hidden/output) is associated with a two way connection with a *Delta-Weight* model. The message sent from the calculation layer model to the *Delta-Weight* contains the data needed to recalculate the weight in order to reduce the output error of the network. On the other hand, the

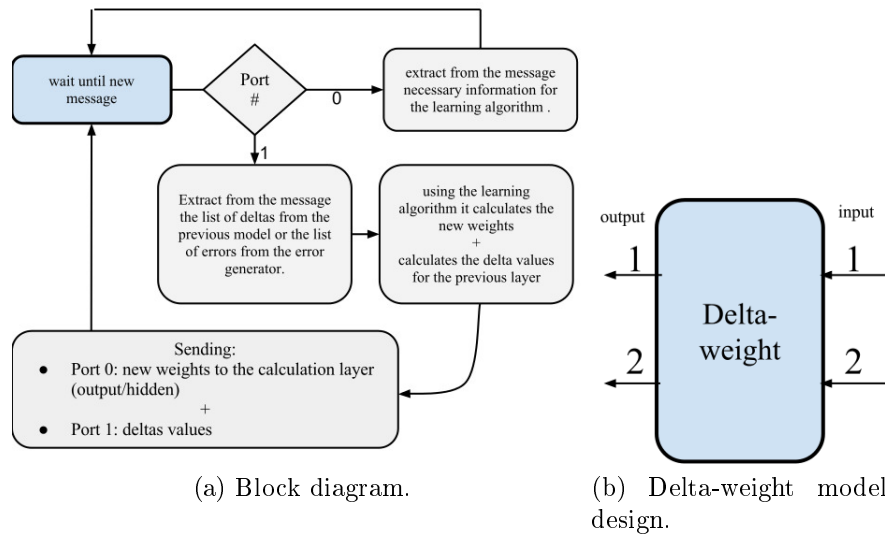


Figure 2.11 – Delta-Weight DEVS atomic model.

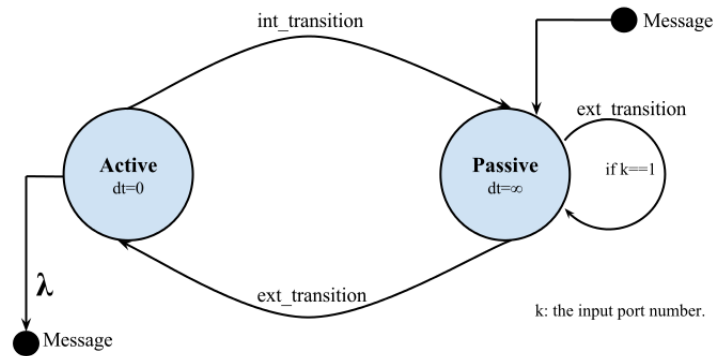


Figure 2.12 – The DEVS state diagram of the Delta-weight DEVS atomic model.

message sent on the other direction contains the new weight list that normally should enhance the ANN performance and reduces the error.

This model always has only four different ports (Figure 2.11b). The *Delta-Weight* has two input ports - the first receives the error list from the *Error-generator*, and the second receives the rest of the information needed for the learning algorithm necessary to recalculate the weight list from the dedicated calculation layer. *Delta-Weight* atomic model sends on port 1 the newly calculated weight list to the dedicated calculation layer, on port 2 it sends the error to the next *Delta-Weight* model.

The *Delta-Weight* model switches between two states depending on which port received the message (Figure 2.12). The model goes through several steps shown by the block-diagram in Figure 2.11a: (I) The model is passive and waits to receive a new message. (II) Once a message is received, there are two pathways the model follows, depending on the port number. (III) If the message arrives on port one,

then it is a message received from a calculation layer that contains the necessary information needed to recalculate a new weight list, and therefore accomplishes the learning process. After that, the *Delta-Weight* model is switched to a passive state. (IV) If another message is received on port two, then it contains the error list that will complete all information need for weight calculations. This message could be sent from a previous *Delta-Weight* or an *Error-Generator* model. (V) Using the back-propagation equations (derivative of the transfer function, error, input, output), a new weight list is calculated for each neuron inside this layer. (VI) After that, the output function sends an output values from both ports: one and two. The first port message contains the new weight list which is sent to the associated calculation layer. The second port message contains the partial derivative error for the previous layer and it is sent to the previous *Delta-Weight* model (see Figure 2.5).

2.4 ANN/DEVS Implementation with DEVSimPy

Nowadays, the Graphical User Interface (GUI) is very essential, as they facilitate the simulation or modeling approaches of any system. DEVSimPy has been chosen for the DEVS-based ANN implementation. After the individual modeling study presented in the previous subsections, the chosen approach is completely implemented as shown in Figure 2.13. In fact, DEVSimPy is the environment that provides the ability to build a neural network model library with several architecture and learning algorithms. This library is the core of any neural network built. This subsection will show the model's implementation as well as a simulation example with one of the first and basic neural network benchmarks, the exclusive OR.

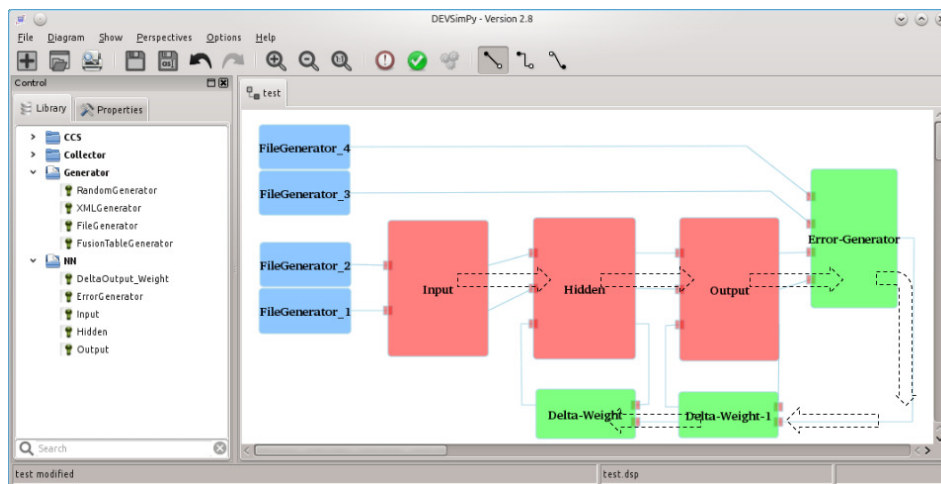


Figure 2.13 – The DEVS-based ANN with DEVSimPy.

The simulation order is shown in Figure 2.13. At first, the *FileGenerators* are active and push messages with necessary patterns to both models: *Input* and *Error-Generator*. The *Error-Generator* collects the desired output data and the *Input* model collects the training and the validation data. After receiving all data from the *FileGenerators*, the *Input* model starts to push messages with the patterns in order to begin the learning process. *Hidden/Output* models continue the feed-forward calculation followed by the learning back-propagation with the *Error-Generator* and *Delta-Weight* models.

On the left hand side of Figure 2.13 the libraries are shown. To create the full DEVS-ANN scheme, we created a full ANN library (Downloadable from: <http://goo.gl/GGpoVV>). This proposed library gives the possibility to extend the neural

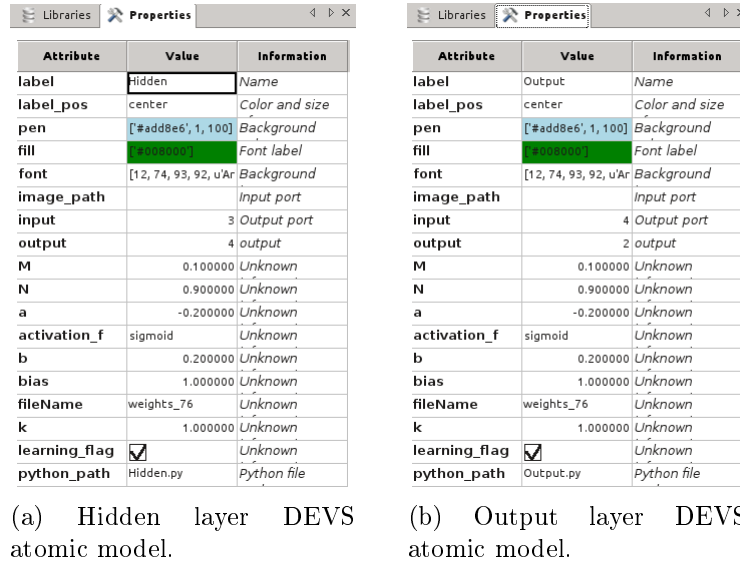


Figure 2.14 – Configuration panels for the DEVS atomic models inside DEVSimPy.

network capabilities with an additional specific model. The *Hidden* and the *Output* models are configurable. The configuration is consisted of the number of neurons inside each layer, the learning factor, the momentum factor and the activation function (Figure 2.14 shows the configuration of the hidden and output models respectively).

Validation

A typical example of non-linealy separable function is the XOR. This function takes two input arguments with values in $\{0,1\}$ and returns one output in $\{0,1\}$, as specified in the following table :

X_1	X_2	O
0	0	1
1	0	0
0	1	0
1	1	1

(a) Set for training.

X_1	X_2	O
0.1	0.1	1
0.9	0.1	0
0.1	0.9	0
0.9	0.9	1

(b) Set for validation.

Table 2.1 – Set of teaching vectors of XOR function.

The input patterns shown by the Table 2.1 are located in file where the *FileGenerator_1* and *FileGenerator_3* can read (Figure 2.13). The Table 2.1b are located where the *FileGenerator_2* and the *FileGenerator_4* can read (Figure 2.13). Both

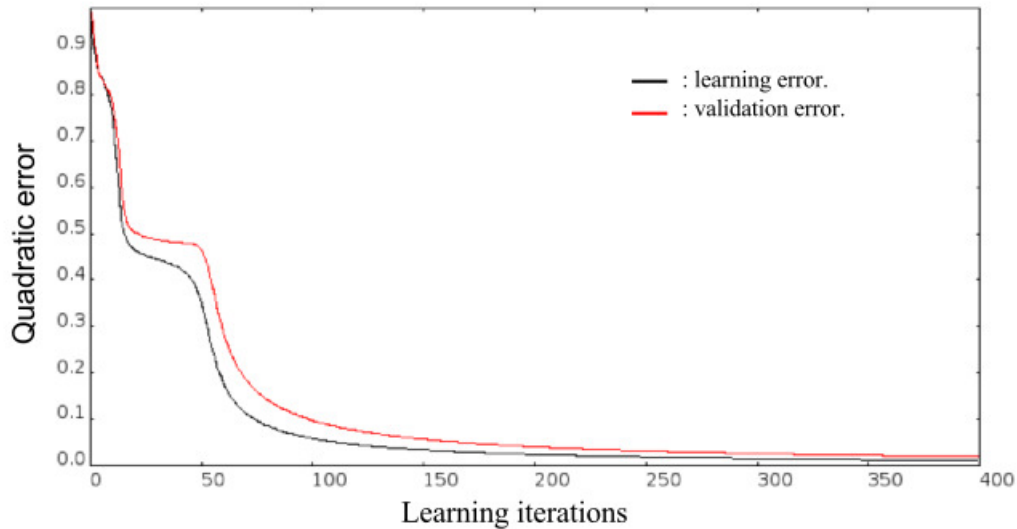


Figure 2.15 – XOR quadratic error DEVSIMPy simulation results.

the *FileGenerator_1* and *FileGenerator_3* send the input and the validation patterns to the *Input* DEVS atomic model. The *FileGenerator_2* and the *FileGenerator_4* send the desired output to the *ErrorGenerator* DEVS atomic model where the learning and the validation errors are calculated.

The exclusive OR function problem is simulated with a standard configuration where one hidden layer with only two neurons, and a sigmoid transfer function. Figure 2.15 shows the quadratic error during the training phase for the learning patterns and the validation patterns too. As shown both the training and the validation curve converge to zero, which means that the error between the desired output and the real output is close to zero. In the case where the learning and the validation curves converges to zero, it means that the ANN was able to map correctly between the input and output of the training and the validation patterns. Also Figure 2.15 appears only when the user double clicks the *Error-Generator* model to visualize the error curve. This is an example of the local atomic model plug-in that overrides the double click event that normally shows the properties panel.

The presented work show three different model categories: the first is the file generators, the second is the feed-forward calculation models set, and the third is the the learning model set. With these three categories, a feed-forward ANN can be built. Inside the DEVSIMPy environment, all models' abilities can be extended with plug-ins implemented on top of the atomic models. This can be used on top of the learning model set in order to observe the learning progress.

2.5 Conclusion

This chapter resumes the brain storming of the artificial neural network modeling approach using the DEVS formalism and the implementation inside the DEVSIMPy environment. Three main designs have been proposed and compared, including the neuron atomic model level, a layer level, and a reduced message layer level modeling. The layer architecture level was chosen to be implemented. This design was presented with two figures: - the first depicting the DEVS atomic models, and the second presents the implementation inside DEVSIMPy environment. This modeling came in order to separate the feed-forward calculations and the learning process. This separation gives the flexibility and the modularity that is necessary to the ANN to ensure the integration of multiple learning and optimizations algorithms. In addition, DEVSIMPy gives the option to add individual atomic model plug-ins, and also global plug-ins that add functionalities and can improve the models performance, and influence their behavior. The DEVS-based ANN library is validated by the simple XOR problem using a single hidden layer model and the Back-propagation algorithm. This integration and the DEVS-modeling of the ANN is the first step towards an optimization and a more flexible ANN configuration.

Chapter 3

DEVS-Based CCS

3.1 Introduction

The concurrent and comparative simulation concept has been used over the last thirty years in several domains. Mainly, it was adopted by the fault simulation of the logic gate level of digital circuits. However, the CCS can be applied to different domains as [26]: the air traffic control, nuclear power plants control, graphs analysis, symbolic simulation, etc. The CCS allows the simulation of a multiple experiments explicitly within one simulation execution.

The core of the CCS algorithms is large enough to allow a simulation process generically for any environment where the system models are built. However, the relationship between the simulation core and the application domain modeling used to be done with a specific modeling interface dedicated to this specific domain.

The advantages of the CCS can be used within the DEVS formalism that explicitly separates the modeling and the simulation approaches. The integration of the CCS with DEVS was developed a few years ago with a simulation core called Behavioral Fault Simulation for DEVS (Behavioral Fault Simulation for DEVS) [16, 74]. The BFS-DEVS is a DEVS simulation core adapted for concurrent fault simulation for discrete event systems. This simulation core was applied and validated on digital circuits and described in the VHDL language. To use the BFS-DEVS simulation core, the user creates a specific concurrent library dedicated to the application domain without being occupied by simulation process, due to the new BFS-DEVS kernel.

The BFS-DEVS extension caught our attention as it integrates the behavioral fault simulation deep inside the DEVS formalism using the CCS algorithms. This

DEVS extension proves the importance, the need and the gain of integrating the CCS algorithms with the DEVS formalism. As DEVS became an interesting formalism, many researchers extended the formalism and added multiple enhancements under the name of DEVS extensions [73, 75, 76]. Figure 3.1, shows the new vision of adding a concurrent behavior to a classic DEVS atomic model. The concurrent behavior will be implemented separately from the modeling approach but with a new linking mechanism able to link the concurrent behavior with one/multiple transition/output function.

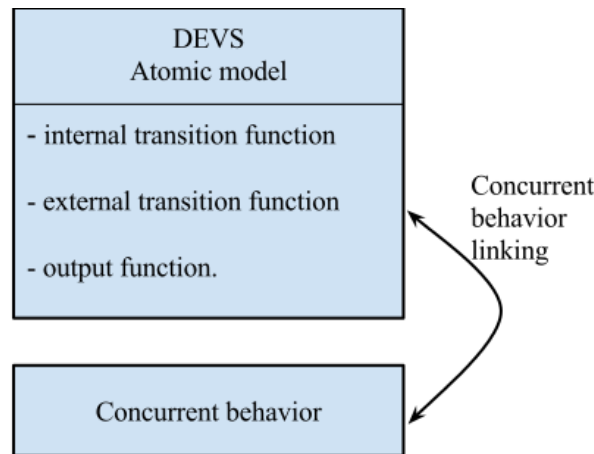


Figure 3.1 – Concurrent behavior linking in the DEVS atomic model.

The approach presented in this thesis is surely inspired by the BFS-DEVS, but there are some major differences. First of all, in our approach, the DEVS classic simulator core is the only simulator used. This way, the presented approach can be applied on top of any DEVS extension. Second, the modeling approach takes into account the two different type of models (concurrent and non-concurrent). Third, the concurrent simulations do not send messages through special ports. All concurrent messages are sent transparently to the user and they are all controlled with one simulation governor. This governor is responsible for adding or deleting experiments on the fly. These three differences can be considered as a generalization of the BFS-DEVS extension.

Figure 3.2 depicts the conceptual framework underlying the DEVS formalism. Four basic objects are the main concern of the modeling and simulation combination. (i) The real *system*, (ii) *model*, (iii) *simulator*, (iv) *experimental frame*. The real *system* is the source of data, in other words: the input/output values at a certain

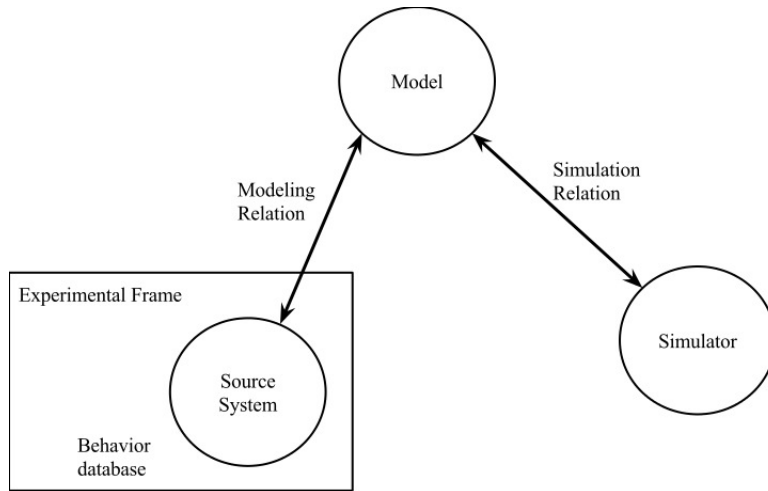


Figure 3.2 – Basic entities and relations in modeling and simulation.

time. The *model* is an informative way to present the structure and the behavior of a system, which are the set of instruction and the set of all possible input/output data that can be generated by the execution of the model instructions respectively. The *simulator* is the execution of the model's instructions to actually simulate its behavior. The *experimental frame* is the specification and the conditions under which the system is observed.

These basic elements are related by two relations: modeling relation and a simulation relation (Figure 3.2). The modeling relation links the real system and the model. A model can be validated when the data generated by the model agrees with the data produced by the real system in a specific experimental frame. The simulation relation links the model and the simulator representing the simulator's ability to execute the model's instructions.

The DEVS-based CCS works mainly on the model entity and its relation with both the system and the simulator. In another words, it changes the modeling in the manner to give the simulator a concurrent behavior without changing the simulator. At the modeling, the implementation of the concurrent behavior is transparent to the simulator making a generalized concurrent behavioral implementation. DEVS-based CCS complements the object oriented programming of DEVS by providing another way of thinking about the simulation structure. It increases modularity by allowing the separation of simulation concerns. The separation between the simulation and the concurrent behavior management without changing the simulator algorithms is a challenging objective. DEVS-based CCS is an aspect-oriented programming (AOP)

solution to avoid multiple concurrent behavior simulators.

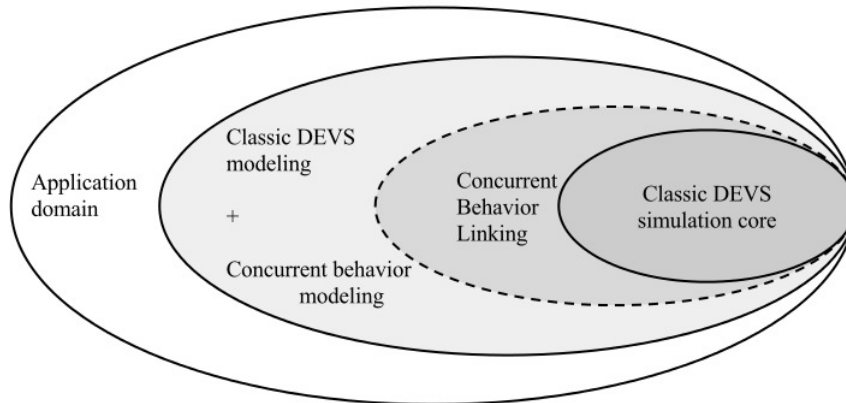


Figure 3.3 – The layers between an application domain and the classic DEVS simulation core.

The relation between the DEVS simulator and the application domain takes place with a modeling interface. Modeling in DEVS formalism has to follow certain rules and constraints, which provides an automatic simulation process. Figure 3.3 shows the relation between the classic DEVS simulator core, the application domain and the introduction of the concurrent behavior modeling. The four circles shown in Figure 3.3 can be divided into two categories: modeling and simulation.

The modeling begins with the study of the application domain, then the creation of a dedicated library of models inside the chosen environment. In our concurrent approach the user might need to add a concurrent behavioral function to the atomic models. Once the user has accomplished the modeling phase, the DEVS-based CCS provides an automatic concurrent simulation by linking the concurrent behavior to the DEVS model with the classic DEVS simulator.

The atomic DEVS model is composed of a set of a functions and variables that represent different behaviors. This modularity in the modeling approach is flexible enough to accept new behavior definitions. The classic DEVS modeling consists of two events: internal and external. The DEVS-based CCS introduces a new concurrent behavior to any DEVS atomic model.

3.2 DEVS-Based Concurrent Simulation

With DEVS formalism the structure of a system is specified using a coupled DEVS model. The coupled models put in evidence the description of hierarchy. On the other hand, the atomic model represents the system behavioral implementation (transition functions, time advance function, and an output function). An atomic model also contains a modeling interface (input and output ports). A DEVS atomic model is represented as a set of component $AM = \langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, t_a \rangle$. This predefined tuples will help the integration of the concurrent behavior without modifying the DEVS simulation core. The concurrent simulation will be applied using only the atomic model behavior.

The first step to realize a concurrent simulation with DEVS is the implementation of a signature manager that handles the concurrent behavior of supported atomic models. The second step is to introduce how an atomic model can adopt the concurrent behavior. Four steps are necessary to this implementation:

- The concurrent behavioral function description and definition inside the DEVS formalism.
- The simulation signature and the concurrent simulation manager that handles the experiments database.
- The CCS algorithm integration inside the DEVS simulator and the execution order for the transition functions.
- The implementation of the DEVS-based CCS into the DEVSImPy environment and test on the artificial neural network library.

The following subsections are going to detail the implementation step by step to get a DEVS-based CCS.

3.2.1 The Concurrent Behavioral Function Modeling

The classic DEVS atomic model represents the behavior of a system. It consists of internal/external ($\delta_{int}, \delta_{ext}$) transition functions, a time advance function (t_a) an output function (λ) a set of input/output ports (X, Y), and a set of states (S). The

time advance function determines the lifespan of a state. The internal transition function (δ_{int}) defines how a state of the system changes internally after the state life time has elapsed. The external transition function (δ_{ext}) defines the how an input event changes the state of the system.

In order to get a CCS simulation, a new transition function will be introduced. The concurrent behavioral function (f_{conc}) is added to the atomic model in order to represent its concurrent behavior. The concurrent behavioral function will be the only function that handles the experiment signatures.

A DEVS experiment is a sequence of atomic models' execution. During a concurrent execution each atomic model can change a portion of the experiment's signature. This portion of the signature is called a **sub-signature**. A sub-signature is the part of the experiment signature private to an atomic model.

A concurrent atomic model has a new complimentary structure described as follows:

$$AM' = \langle X, Y, S, \{H_n\}, \delta_{int}, \delta_{ext}, f_{conc}, \lambda, t_a \rangle$$

Where:

- $X = \{(p, v) | p \in in_ports, v \in X_p\}$ - the set of inputs ports and values,
- $Y = \{(p, v) | p \in out_ports, v \in Y_p\}$ - the set of output ports and values,
- S is the set of state variables,
- H_n : is the set of sub-signatures where:
 - $H_n = \{(X_n, Y_n, v_n)\}$,
 - Where:
 - * X_n is the set of inputs ports and values for an experiment n ,
 - * Y_n is the set of output ports and values for an experiment n ,
 - * v_n is the set of private variables for an experiment n ,
 - * $n \in \mathbb{N}$ where $0 \leq n \leq N$, N is the number of experiments per simulation.
- $\delta_{int} : S \rightarrow S$ - the internal transition function,

- $\delta_{ext} : Q \times X \rightarrow S$ - the external transition function where,
 - $Q = \{(s, e) \mid s \in S, 0 \leq e \leq t_a(s)\}$ - the set of states,
 - e is the time elapsed since the last transition.
- $f_{conc} : H_n \rightarrow H_n$ - the concurrent behavioral function,
- $\lambda : S \rightarrow Y$ - the output function,
- $t_a : S \rightarrow \mathbb{R}_{0,+\infty}^+$ - the lifetime of the state S , $t_a \in [0, \infty[$.

Unlike the BFS-DEVS and its faulty function that is executed after the external transition function for fault simulation, the concurrent behavioral function is capable of replicating different behaviors depending on the model's type. For a generator behavior atomic models (no input ports and a useless external function) the user links the f_{conc} to the internal transition function behavior. The f_{conc} is the only function capable of changing the concurrent simulation signature (or sub-signature) called H_n but has no control over the attributes of the classic atomic model. This means that the presence of the concurrent function does not affect the behavior of the classic atomic model. This behavior comes to confirm the optional presence of the transition function for the classic atomic model simulation. However, the concurrent behavioral function can only handle the experiments sub-signatures without any other access to the classic DEVS variables. Figure 3.4 shows that an atomic model can access the experiments' sub-signatures via a single point of access through the concurrent behavioral function. Each atomic model has one private sub-signature per experiment. For N experiments, an atomic model has to have N sub-signatures.

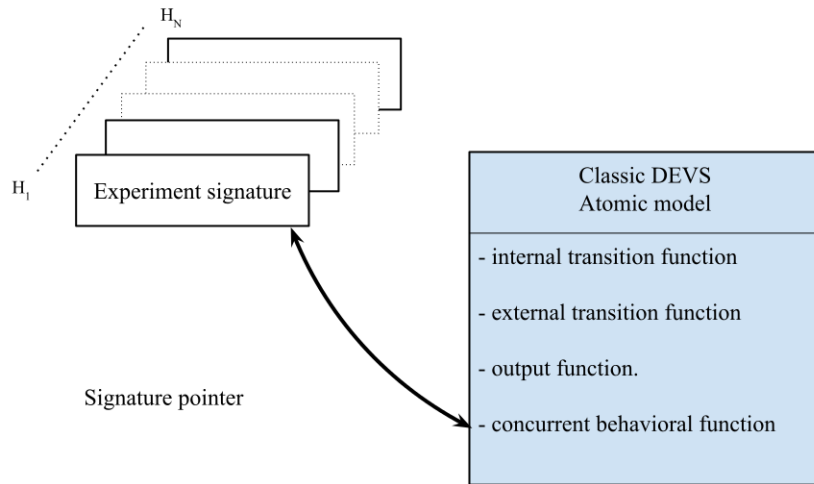


Figure 3.4 – A single atomic concurrent DEVS model with a signature pointer.

Experiment signatures and sub-signatures are all stored in one place, the **singleton signature database**. The idea behind gathering all signatures in one place is to make it easier for simulation managing.

3.2.2 Singleton Signature Database

The classic DEVS formalism with ports has a set of inputs X . This set of inputs, as well as the set of outputs Y and some other variables are considered as part of an experiment **sub-signature** for one atomic model. The experiment signature is only considered for a DEVS-based concurrent simulation. The signature definition takes part of the modeling process that the user has to accomplish. Each user, while modeling the chosen system, has to define the value that will compose the signature of an experiment. In fact, any private variable that changes from one experiment to another is considered as part of the model sub-signature and therefore the experiment signature.

Figure 3.5 explains an important concept for the implementation of the concurrent and comparative simulation using the DEVS formalism. The concept is based on Figure 3.4 where each atomic model individually accesses his own sub-signatures. All signatures have to be stored in one place for easy management and more control. Regardless of the implementation of the signature manager, it will always be easier to manage all experiments in one place. In other words, one instance with a single point of access is more controllable than multiple distributed ones.

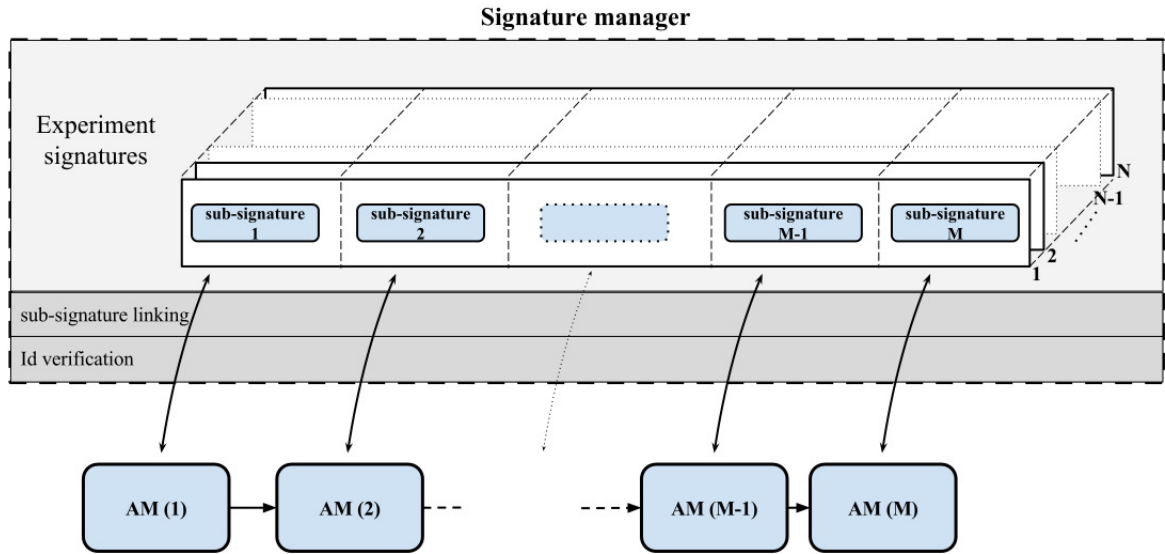


Figure 3.5 – Singleton signature database.

Restricting the class to *one instance* guarantees the role of coordinator between concurrent operations. Models can only access the simulation data (sub-signatures) by entering the singleton design pattern. Figure 3.5 shows that atomic models access the singleton class as if it is the signatures database. Atomic models can only reach their own sub-signature(s) and do not have the permission to modify other sub-signatures. This permission is provided through the signature manager by checking the sub-signature ID. The concurrent behavioral function provides the atomic model ID to the signature manager that responds with the experiments sub-signatures dedicated to this model. As an example the atomic model (1) presented in Figure 3.5 has access to its own sub-signatures inside each experiment object. Each atomic model m has access to all sub-signatures with ID n for each signature m , as $0 < n \leq N$ and $0 < m < M$, where M is the number of the atomic model and N is the number of experiments executed. The database handles all signatures in one place, making it easier to manage experiments and compare between them.

Providing a *global point of access* to all atomic models can offer a better multi-experiment management. The signature manager is the only source to provide atomic models with signatures. Each atomic model presents its own ID to the signature manager in order to get access to data (signature and sub-signatures). At that point the signature manager gives access to the atomic model to the data with the given permission (read and/or write). In that manner, any experiment deleted/added with

the signature manager will not/will be accessible for any atomic model. Regardless the reason for deleting or adding an experiment, doing so in one central instance accessed by all atomic models is much easier than doing it individually with every atomic model.

The previous two concepts: *one instance* and *global point of access* lead to a well-know concept in design patterns for software engineering [77]. Design patterns are a general reusable solution to a commonly occurring problem within a given context in software design. The proposed solution (the singleton database) is categorized under the creational patterns. The **singleton pattern** is a design pattern that is useful when exactly one object is needed to coordinate actions across the system. The singleton pattern will be applied in the form of a signature manager class. In that manner, it provides sub-signature access to all atomic models and manages only one instance to compare/add/delete experiments within the same simulation.

The singleton class, or in other words the singleton manger has a list of the running experiments. Each experiment has a list of models sub-signatures (see Figure 3.5). This signature hierarchy facilitates the adding or deleting of new experiments. When the signature manager deletes an experiment, all sub-signatures are deleted consequently.

3.2.3 The Concurrent Behavioral Function Simulation

An atomic model allows specifying the behavior of a basic element of a given system. An atomic DEVS model can be considered as an automaton with a set of states and transition functions allowing the state change when an event occurs or not. When no events occurs, the state of the atomic model can be changed by an internal transition function called δ_{int} . When an external event occurs, the atomic model can intercept it and change its state by applying an external transition function called δ_{ext} . The life time of a state is determined by a time advance function called t_a . Each state change can produce output message via an output function called λ . A simulator is associated with the DEVS formalism in order to exercise instructions of coupled model to actually generate its behavior. The architecture of a DEVS simulation system is derived from the abstract simulator concepts associated with the hierarchical and modular DEVS formalism.

The DEVS formalism has a classic simulation core that attributes coupled models to coordinators and atomic models to simulators (chapter 1.3.2). Coordinators are responsible for the time management and control of the message exchange between simulators. Coordinators works in accordance with the coupled model specification. Simulators execute the atomic models and generate their behavior. Classic DEVS

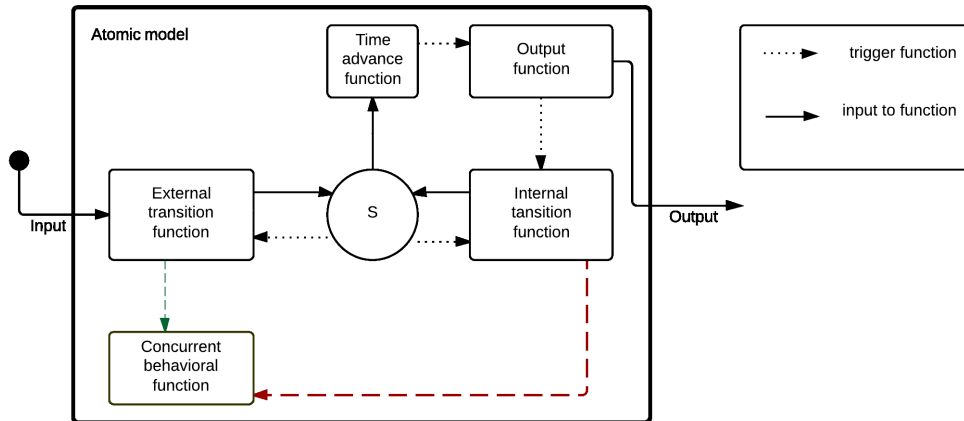


Figure 3.6 – The concurrent DEVS atomic model simulation.

simulators consider two transition functions: internal and external functions. The DEVS-based CCS goal is to extend atomic model functionality dynamically (at run-time) to be able to get a concurrent behavior. Our intent is to add additional responsibility to the atomic model object at run-time. At the same time, it has to keep the same interface to be executed normally with the DEVS classic simulator. Decorating objects (models) functions might be more convenient than changing the entire simulator algorithm each time the user needs to extend the model behavior.

Figure 3.6 shows the concurrent DEVS atomic model simulation interpretation. At any time the system is in some state s . If no external event occurs the system will stay in state s for time $t_a(s)$. The concurrent concept can be realized by triggering the concurrent behavioral function after any of transition functions of an atomic model (Figure 3.6). Subsequently, this means adding concurrent behavior to the atomic model through a procedure known as the concurrent behavioral linking.

In the first case where the t_a is so short that no external event can intervene, the

internal transition function is executed changing the model state from s to a new state s' and trigger the concurrent behavioral function - if the concurrent behavioral function is linked to the δ_{int} . This concurrent function just dialogue with the signature manager in order to propagate the concurrent behavior through the system.

In the second case, the system will stay in s forever unless an external event interrupts its passivity. If an external event occurs, the system changes to a new state s'' . If the concurrent behavioral function was associated with the δ_{ext} then it was triggered just after the δ_{ext} was executed.

When the resting time expires, i.e, when the elapsed time, $e = t_a(s)$, the system outputs the value, $\lambda(s)$, and changes the state $\delta_{int}(s)$. Note output is only possible just before internal transitions.

The concurrent behavioral linking is a procedure to adapt a concurrent model in order to respect the DEVS classic simulator interface. The linking procedure has to link the concurrent behavior to one of the two internal or external transition functions. During the linking process, the atomic model gains access through the signature manager to the signature database to be used in the concurrent behavioral function.

3.3 DEVSimPy Implementation

The DEVSimPy environment has implemented the DEVS atomic/coupled models based on an object-oriented programming paradigm. On the other hand, the DEVS-based CCS is the aspect-oriented programming model that provides a new way of thinking about concurrent and comparative simulation algorithm. This subsection shows an implementation of the DEVS-based CCS inside the DEVSimPy environment. All implementations are in python programming language. The dynamic Python programming language - used as scripting language that helps the *on the run* code modification - is used for the implementation of the DEVSimPy environment. The dynamic feature is helpful for the DEVS-based CCS concept during the linking process as the concept needs to modify models on the fly to integrate the concurrent behavior in order to match the classic DEVS simulator. The implementation takes several steps: (i) the creation of the signature structure, (ii) the implementation of the concurrent behavior function, (iii) linking the concurrent behavior to a transition function, (iv) the simulation managing, (v) as well as the signature manager extensibility using a DEVSimPy plug-in.

DEVS atomic models have a predefined structure to follow as well as a set of private variables that can be considered as the model configuration. Encapsulating all the simulation-dependent variables inside one generic object is a practical solution for a dynamic structure. This dynamic structure is called the model sub-signature. An atomic model signature is an object that contains private variables that can be changed during the simulation. Figure 3.7 shows the UML diagram for the implementation of the DEVS-based CCS using different classes and interfaces.

3.3.1 Classes Description

The UML diagram (Figure 3.7) can be categorized into three main groups: (i) the atomic model set, (ii) the signature set, (iii) the manager set. The atomic model set is composed of the *ConcreteAtomicModel*, *<interface>AtomicModel*, *DecoratorAtomicModel* and the *ConcurrentAtomic*. This set is the class diagram representation to create a decorator for the classic DEVS atomic model in order to get a concurrent runtime model. The signature set is composed of the *Experiment* and the *Signa-*

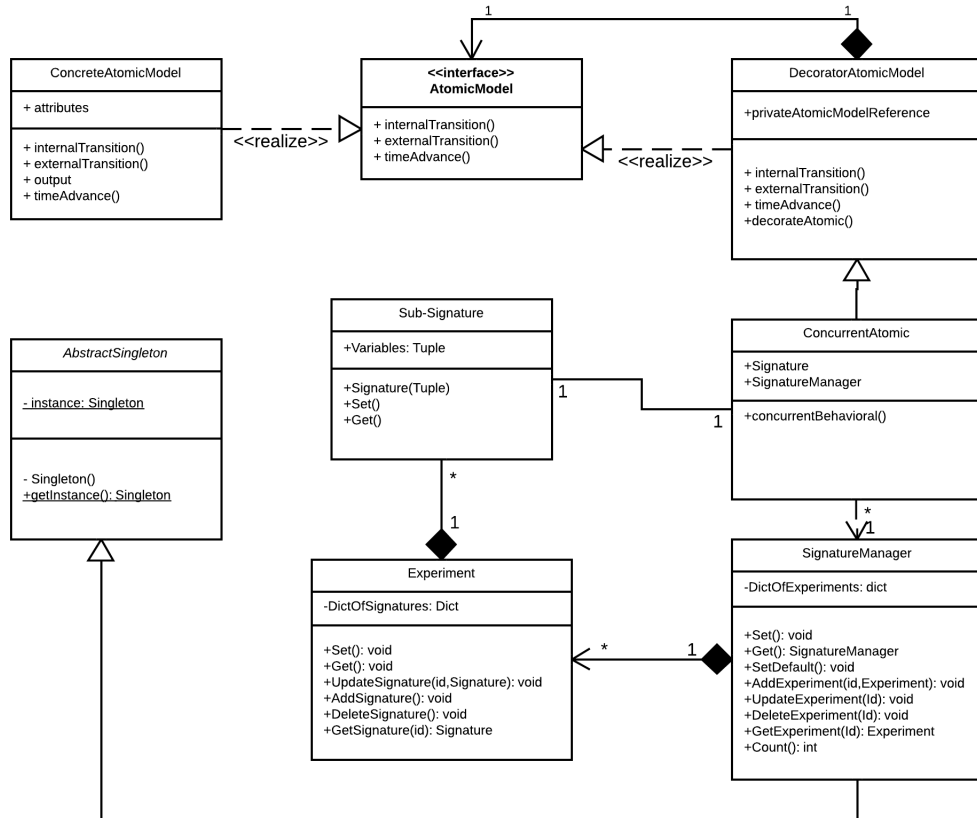


Figure 3.7 – DEVS-based CCS UML class diagram.

ture classes where each experiment is composed of many signature (sometimes are called sub-signatures) and a signature only belongs to one experiment. The manager set is represented by the *SignatureManager* and the *AbstractSingleton* which are the database and where the stack of experiments is handled, updated and compared to each other.

In software engineering the decorator is a structural pattern. This pattern offers a standard implementation to functionality extension at run-time. Figure 3.7 contains the UML description of the decorator pattern. The UML notation contains an atomic model interface, which is the basic function needed for the classic DEVS simulator. The concrete atomic model defines an object to which additional responsibilities can be added. The decorator class maintains a reference to a component object and conserves the interface of the classic atomic model. The concurrent atomic model

extends the functionality of the component by adding the signature state and the concurrent behavior. The atomic model decorator attaches the concurrent behavior by default to the external transition function. It is important to note that the concurrent transition is always executed after the transition function.

The SignatureManager class (Figure 3.7) implements the abstract singleton class that has one static instance of the class *Singleton* and one static function called *getInstance()*. The signature manager returns the same instance each time the *getInstance()* function is called. The signature manager class provides some implemented functions to help managing different experiments and update the database with the signatures modifications during the simulation time. A *setter* and *getter* are implemented to protect the access to the singleton instance. *setDefault()* is an optional function that can be used to set the default configurations for a specific experiment. This function has to access all the concurrent atomic models signatures to be able to set the default configurations.

AddExperiment(id, Experiment) and *DeleteExperiment(id)* are two methods that are used by the DEVSimPy plug-in functions to add and delete experiments based on user request or any implemented automatic algorithm. Also the plug-in functions are responsible for generating new IDs for the new experiments. The *AddExperiment(id, Experiment)* methods receives the new ID and an experiment object. If the received ID already exists, the *UpdateExperiment(id, Experiment)* is called in order to update the current experiment with the new data. If the received ID is new, a new experiment will be stored in the experiments dictionary with the attached ID. The *Count()* function returns the number of the simulated experiments at any time of the simulation. The signature manager can handle multiple experiments at a time, but also if the signature manager is deleted there will no longer be any experiment.

The Experiment class (Figure 3.7) has some similarities with the signature manager, both have to manage a dictionary of objects and be able to update, delete and add new objects to it. The experiment object is only instantiated in the signature manager class, as it represents a concurrent experiment. The Experiment class contains the sub-signature of all the concurrent atomic model in the system. The experiment class has as many sub-signatures as the number of concurrent atomic model in the system. If the experiment is deleted then all of the sub-signatures inside it will

disappear too. It can only have one signature manager.

The sub-signature class is a dynamic structure that offers an encapsulation to the atomic model variables that can change during the simulation. With the DEVS classic atomic models, variable declaration is a user decision, he can create his own conventions. With a concurrent atomic model, it is recommended to use the signature structure to optimize the performance of the concurrent behavior. In that way comparison between experiments can be much easier and generic. The constructor of this class receives a dictionary, where the keys represent the variables' name and the values represent the variables' values. It creates an object containing all necessary variables for the concurrent atomic model in a unique experiment. In the case when simulating multiple experiments, the concurrent atomic model can have multiple sub-signatures as previously explained with Figure 3.5. Each sub-signature instance is attached to only

The last three classes (**SignatureManager**, **Experiment**, **Sub-signature**) all are standalone classes. They are the implementers of the base structure to begin any dynamic simulation changes. In other words, the Signature class is the small unit that presents the fingerprint of an atomic model at a time t during the simulation. The Experiment class is the encapsulation of all concurrent atomic model signatures, and can be used for comparing between multiple experiments at a chosen time t . The **SignatureManager** class is the database where experiments, signatures and sub-signatures are stored and is a single point of access for all models to modify their fingerprints (signatures). The next subsection describes the dynamic part of the approach, where the linking between the DEVS atomic model and the previous implemented classes using a DEVSimPy plug-in.

3.3.2 DEVSimPy Global Plug-in Integration Functions

The DEVSimPy environment offers the possibility of implementing global plug-ins (Figure 3.8). Any user can implement a custom plug-in while respecting the registry interface. The programmer has to import the plug-in manager package and then register it by coding: `@pluginmanager.register("function_name")`. The DEVS-based CCS requires a plug-in called **CCS manager** which is implemented and has a start registered function called: `start_concurrent_simulation(*args, **kwargs)`. After that,

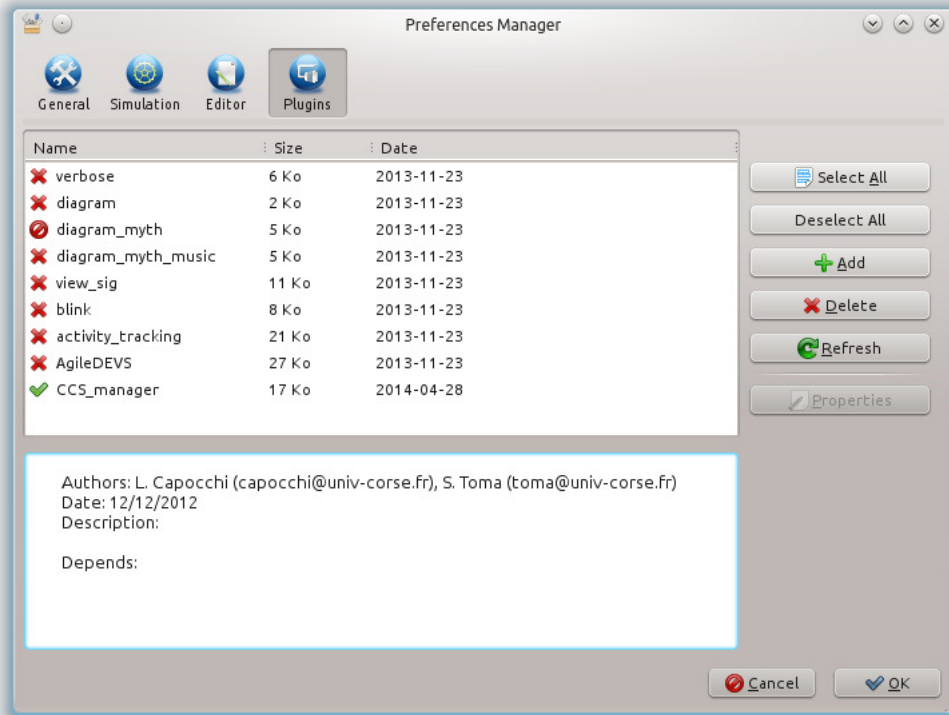


Figure 3.8 – DEVSIMPy plug-in manger interface.

the plug-in shows up in the plug-in manager interface (Figure 3.8). The plug-in has to be activated before the simulation starts to take effect. The DEVS-based CCS the *start_concurrent_simulation(*args, **kwargs)* is executed by DEVSIMPy just before the DEVS simulation starts.

The CCS manager plug-in collects a list of all concurrent atomic models (models that implement a concurrent behavioral function). After the experiments' configurations and before the simulation run-time, the CCS manager actually converts atomic models with an implemented concurrent behavioral function into a real concurrent atomic model that runs under any classic DEVS simulator.

The decoration of the transition functions by the concurrent behavioral function is presented by the following code (Algorithm3.1):

This code shows that the decoration process is by default applied to the DEVS atomic model external transition function (Line number 1, Algorithm 3.1). After the decoration process on either of external or internal transition function the function returns a new concurrent DEVS model able to communicate with the CCS manager.

The concurrent configuration panel appears when the user clicks on the properties

Algorithme 3.1 Concurrent decoration function.

```

1 def concurrent_decorator(model,func='extTransition'):
2 ''' Atomic model decorator for the concurrent behavior '''
3 if func == 'extTransition':
4     model.extTransition = decorator(model.extTransition)
5 elif func == 'intTransition':
6     model.intTransition = decorator(model.intTransition)
7 return model

```

button shown on the left hand side of DEVSimPy plug-in manager. This panel is composed of two tabs: model and simulation.

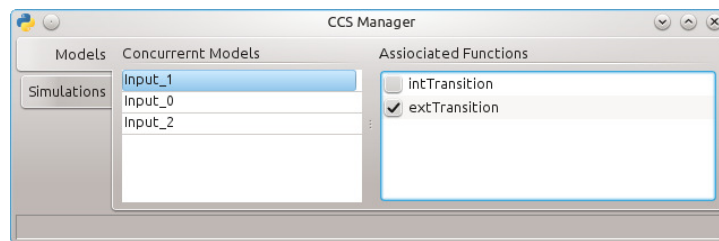


Figure 3.9 – Model association panel.

The **model tab** (Figure 3.9) allows the user to choose for each atomic model the transition function that will be linked to the concurrent behavioral function. By default, the external transition function is checked. The CCS manager is capable of associating the concurrent behavior to both internal and external transition functions by checking both functions in the list.

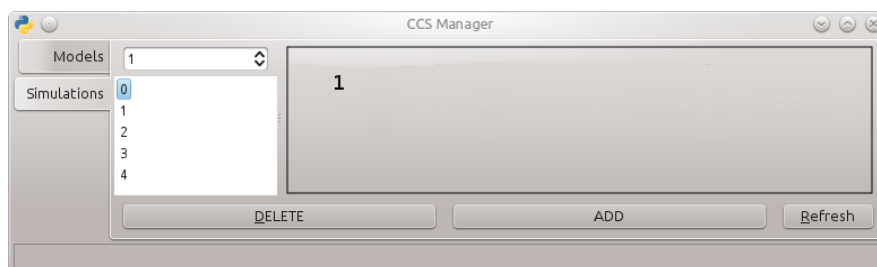


Figure 3.10 – Simulation configuration panel.

The **simulation tab** contains a customizable panel which can change depending on the application. The user can develop his own configuration panel (Figure 3.10, Zone 1) depending on the atomic models' configurable signatures. This panel is configurable, and the user can make his own interface using wxFormBuilder [78]. This panel is useful in order to change at runtime any configuration needed to a

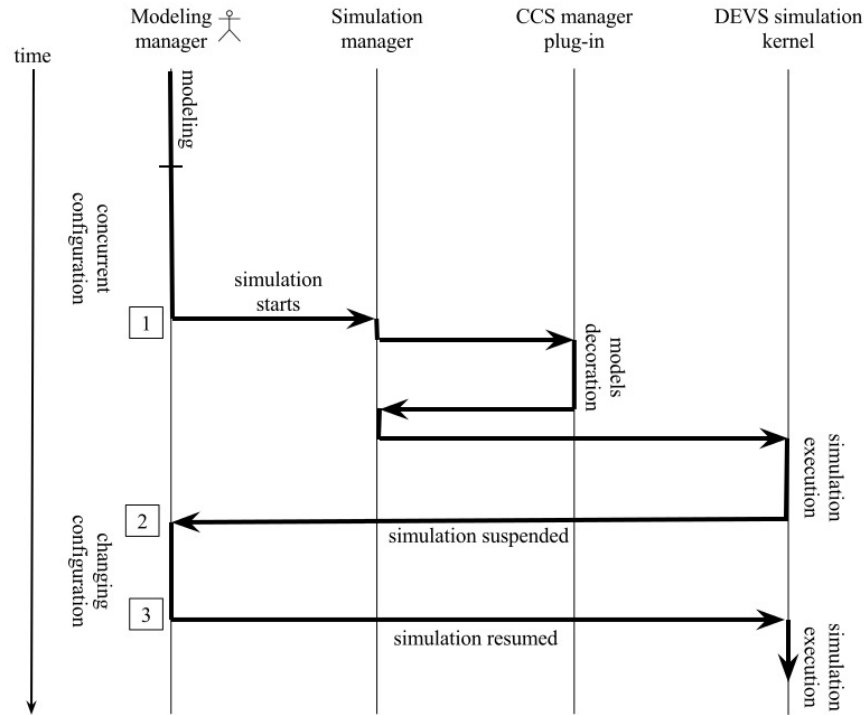


Figure 3.11 – Sequence diagram for the concurrent simulation scenario inside DEVSimPy.

specific experiment chosen from the list on the right hand side of the CCS manager interface (Figure 3.10).

3.3.3 Modeling and Simulation Sequence Diagram

The DEVSimPy environment offers a modeling and a simulation sequence that enables the integration of the plug-ins in a transparent way to the modeling manager (the user) and the simulation kernel. As shown in Figure 3.11 the user only interacts with the system as the modeling manager in steps 1,2 and 3 of the figure. The sequence diagram shown in Figure 3.11 can be decomposed into several steps to explain the relation between the user the simulation kernel in order to perform a concurrent simulation:

1. The DEVSimPy modeling manager is where the user interacts with the environment to create his system using DEVS atomic/coupled models. At user can either use models from the libraries or create new models.
2. Adding the concurrent behavior and configuration comes in the modeling phase where the concurrent behavioral function is implemented.

3. After the modeling and the concurrent configuration, the user can click on the simulation panel (see Chapter 1, Figure 1.15) to start the simulation process (step 1 in Figure 3.11).
4. The DEVSimPy simulation manager takes the relay from the modeling manager to start the simulation process. The first thing done by the simulation manager is to check for the active global plug-ins and integrates their role.
5. The CCS manager is the necessary plug-in to activate the concurrent behavior for the DEVS simulation. For that reason it decorates all of the concurrent models via the decoration function (Algorithm 3.1), making the concurrent behavior transparent to the DEVS simulation kernel.
6. The simulation manager takes executes the DEVS simulation kernel after the CCS manager plug-in has done his job.
7. At any moment of the simulation the user can suspend the simulation to change any model configuration. This is done by clicking on the suspend button on the simulation panel in Figure 1.15.
8. After changing the configuration using the DEVSimPy modeling manager, the user is able to resume the simulation in order to get the desired result.

Two main points for the DEVS-based CCS are identified with previous sequence diagram: (i) the user only interacts with the simulation process only in step 1, 2 and 3 in Figure 3.11 making the concurrent simulation process completely transparent for him, (ii) The concurrent simulation is completely transparent to the DEVS simulation kernel too, making it easy to change the simulation architecture at any time without affecting the concurrent behavior.

3.3.4 Validation: PC Model

To illustrate the concurrent concept with an example, the Computer example shown in Chapter 1.3.2 will be presented with the concurrent behavior. Only the *PC* coupled model will be presented as shown in Figure 3.12. The *PC* is a coupled model composed of one queue (*RQ* atomic model) and one processor (*Proc* atomic model). The role

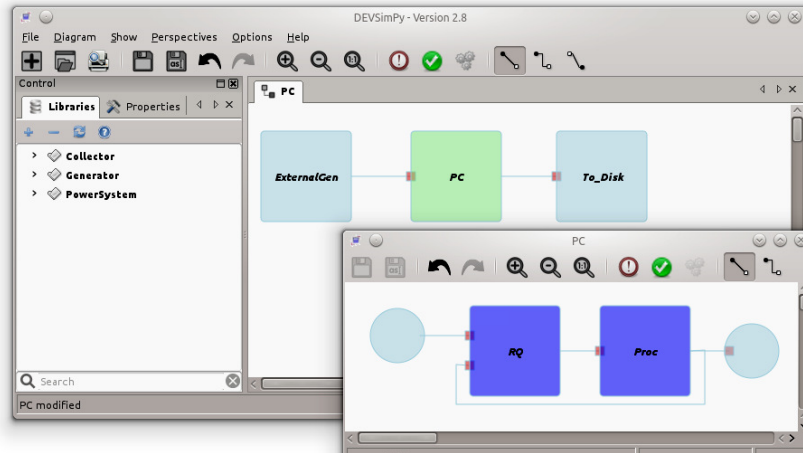


Figure 3.12 – Concurrent PC DEVS coupled model for concurrent simulation.

of the processor is to calculate the sum of each value sent from the RQ model. All implementations are done with DEVSimPy on a Linux, Kubuntu 12.04 distribution with the KDE Layer. Kate is the main code editor used while developing the DEVS-based CCS classes and plug-in. Some of the graphical interfaces are implemented with wxFormBuilder [78] to build XRC extension files, providing a very generic solution to any user who wants to edit or modify the GUI.

Algorithm 3.2 Concurrent behavioral function for the CPU atomic model.

```

1 def concBehavFunc(self):
2     if self.state['status'] == 'ACTIVE':
3         Singleton = self.simData.getInstance()
4         for exp in self.expDico:
5             subSignature = Singleton[exp][self.myID]:
6             subSignature['outputs'] = subSignature['inputs']
7                                     + subSignature['Sum']

```

The Algorithm 3.2 shows the code of the concurrent behavioral function for the CPU model where the basic function of calculating the sum value of all inputs. This shows the concurrent function retrieving and saving the data from the simulation manager in order to perform the concurrent behavior for each experiment.

Scenario: The RQ model reads a file with a randomly generated numbers n where $n \in [0, 10]$. When the $Proc$ model receives a value it takes a time $t = 1sec$ to perform the calculation and generates an output. After calculation, the $Proc$ model generate the output and sends a signal to the RQ model to send the next values. This example will use the concurrent simulation in order to calculate the sum of five different files.

In a normal DEVS simulation, the user should perform five different simulations with exactly the same steps in order to get the results. With the DEVS-based CCS the user has only to configure the concurrent behavioral function and enable the CCS manager to perform the concurrent simulation. The input files are shown in Table 3.1a, where File1, File2, ..., File5 are the different files read by the *ExternalGen* model and sent to the *PC* coupled model. The output of the *PC* coupled model is read by the *To_Disk* model that is only a graphical representation for data. The *To_Disk* overrides the double click function of the atomic model with a local plug-in and displays instead the data in a form of a table as shown in Table 3.1b.

File1	File2	File3	File4	File5	Time t	O1	O2	O3	O4	O5
1	2	3	6	9	1	1	2	3	6	9
3	3	2	4	7	2	4	5	5	10	16
5	4	1	2	6	3	9	9	6	12	22
2	5	3	2	4	4	11	14	9	14	26
4	3	2	1	3	5	15	17	11	15	29
6	6	4	3	2	6	21	23	15	18	31
7	3	2	4	5	7	28	26	17	22	36
9	2	1	5	6	8	37	28	18	27	42
1	1	1	6	6	9	38	29	19	33	48

(a) The input files.

(b) The output data.

Table 3.1 – The input and output data of the concurrent PC system.

This is a very example of a system that can benefit of the DEVS-Based CCS to simulate and compare different experiments within a single execution. Until now the DEVS-based CCS works perfectly with a system that has a single simulation path with different values and configuration.

3.4 Conclusion

DEVS-based CCS is a concept that merges the concurrent and comparative simulation algorithms with the classic DEVS simulator's efficiency and modularity. The **DEVS-based CCS** works on disguising the concurrent DEVS atomic model into a classic atomic model able to be executed with any standard DEVS simulator. It is a generalization of the concurrent DEVS extensions, where the DEVS functionality is extended without changing the simulator core. This concept works on decorating the transition functions with the concurrent behavior depending on the nature of the atomic model. For a passive model, the external transition function is decorated with a concurrent behavior. For active models, the internal function is the one to be decorated. For some particular applications, both internal and external transition functions might need to be decorated.

The implementation of this concept is presented inside the DEVSimPy environment. Any atomic model can have a concurrent behavior by the presence of a *concurrent behavioral function* and a *signature* object. CCS manager is a DEVSimPy plug-in that decorates one or both transition functions with the concurrent behavioral function and manages the experiment's sub-signatures. The plug-in has an interface that controls the model's configuration and the experiment's branching and merging.

The DEVS-based CCS is fully implemented and validated with an example of the *PC* coupled model. The **concurrent DEVS-based neural network** is the core of the new wound-rotor induction generator short-circuit diagnosis based on digital data. The next chapter goes into details to add the concurrent behavior for the DEVS-based ANN as well as a classification optimization.

Chapter 4

Concurrent DEVS-based ANN

4.1 Introduction

This chapter regroups the two concepts introduced in Chapters 2 and 3 (DEVS-based ANN and the DEVS-based CCS). First in Chapter 2, due to the modularity of DEVS, the DEVS-based ANN introduces a fragmentation of the Feed-forward ANN architecture in order to decompose the learning phase from the feed-forward calculations creating a new ANN DEVS library. Chapter 3 introduces a transparent atomic model modification to the DEVS simulation kernel in order to deliver the concurrent behavior. This modification can be applied to a system with a single simulation path but with different data and configurations. The ANN has various possible configurations where all need to be simulated in order to pick and choose the best configuration possible for a specific application [25]. This multiple configuration problem makes the DEVS-based ANN very beneficial from the concurrent behavior.

The main concepts of the DEVS-based CCS are validated with the DEVS-based ANN, which is a single path simulation. By a single path we mean that the atomic models are activated in a specific order regardless the experiment or the configuration.

The concurrent behavior for the DEVS-based ANN is presented with details in this chapter. It begins with the concurrent behavioral function design and implementation, followed by the concurrent simulation configuration and results visualization. After the validation of the concurrent behavior with the ANN, a classification optimization based on some statistical calculation is proposed for the concurrent DEVS-based ANN.

4.2 Adding the Concurrent Behavior to DEVS

The DEVS-based CCS was first implemented inside the DEVSimPy environment in a general form, which was later adapted to a special application. This section represents the adaption steps of the DEVS-based CCS on the DEVS-based ANN. Firstly, the atomic model has to be adapted for the concurrent behavior takes place. Secondly, the CCS manager has to be personalized and adapted to handle the ANN configuration needs. For example the ANN configuration has to be changeable through the CCS manager.

Figure 4.1 shows the interaction between the basic atomic model and the signature manager. Each atomic model can gain different access permissions to the experiment signatures and model sub-signatures. The two permissions that can be given to a concurrent atomic model **are the read** and **write** permissions.

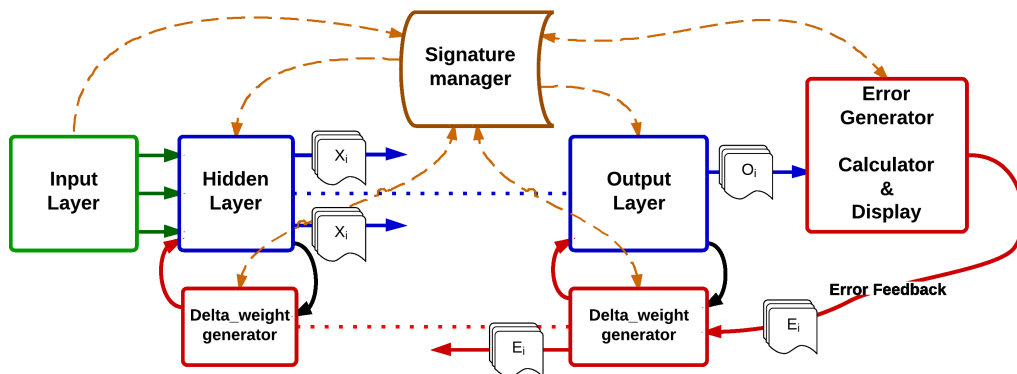


Figure 4.1 – DEVS-based ANN with a concurrent learning behavior.

Figure 4.1 elucidates the models' permissions using directional arrows. Every atomic model is given permissions based on different purposes:

- The ***Input*** layer only has the writing permission to be able to add for every experiment the needed input to continue the calculations. It doesn't need to read any other information.
- The ***Hidden/Output*** layers have both; reading and writing permissions to be able to read the inputs and write back the outputs.

- The *ErrorGenerator* models have to read the output of the ANN, calculate the error during the training phase and compare between the different experiments. Consequently, the model writes back the error compared to the ideal output desired by the user.
- The *Delta_Weight* generators models read the errors calculated by the *ErrorGenerator* as well as the weights of the associated output/hidden layer model. The write permission is given allow the modification of the weight list for each experiment.

The read/write permissions are given by the CCS manager during the decoration process when the atomic model gets the signature reference. Each atomic model needs the concurrent behavioral function to be implemented in order to take into consideration the concurrent simulation explained in Chapter 3.

Figure 4.1 is the best way to explain the concurrent simulation alongside of the classic simulation. At all times, the classic DEVS simulation with the conventional messaging protocol is always there. On the other hand, the concurrent experiments simulation exchanges the necessary messages only through the signature manager. As shown, the input layer atomic model sends messages to the hidden layers models followed by the output, error generator, etc. Both messaging techniques are used simultaneously, one for the classic DEVS simulation and the other for the concurrent experiments simulation.

Algorithm 4.1 Concurrent behavioral function for the Input atomic model.

```
1 def concBehavFunc(self):
2     if self.state['status'] == 'ACTIVE':
3         Singleton = self.simData.getInstance()
4         for exp in self.expDico:
5             if not self.myID in Singleton[exp]:
6                 e = self.createExp()
7                 Singleton.AddExperiment(exp, self.myID, e)
```

The *Input* layer model, can only add new sub-signatures to the signature database, unlike all the other models that can read/write to the signature database. Algorithm 4.1 is a piece of code of the concurrent behavioral function for the *Input* layer atomic model. The "*concBehavFunc*" function tests the state of the model. If "*ACTIVE*"

(Line 2, Algorithm 4.1), then it checks the signature manager (the Singleton) if the input layer signature already exists, if it is not the case a new model sub-signature is added to the singleton class.

The created ANN library in Chapter 3 was designed for the classic DEVS simulator. At this stage, the library is modified to be able to work on the classic DEVS simulator with the classic behavior and with the concurrent and comparative behavior using the CCS plug-in. The CCS plug-in adds a new functionality to the DEVS formalism transparently to the simulator and the user. With model changing and a customized configuration panel, the next sub-section presents a validation test on the N-parity problem that also introduced to validate the classic DEVS-based ANN.

N-parity problem

Description

The N-parity problem is one of the first non-linear separation problems that validated the capability of the back-propagation learning algorithm to solve complex problems. It was also used in Chapter 2 in order to validate the DEVS-based ANN fragmentation between the learning algorithm and the feed-forward calculations. To make the tests more challenging and interesting, we will use the 4-bit parity problem, which means a 4-bit problem differencing between two main classes as shown by Table 4.1. In this table, $x_1 - x_4$ are the 4 input bits of the ANN and the desired output is presented by the y .

Modeling and Simulation

Two types of configurations will be tested in this example. First, the number of neurons inside the Hidden layer (Table 4.2a), and how it can affect the simulation. Second, is the learning (N) and momentum factor (M) (Table 4.2b) which can affect the learning speed and the learning curve convergence. As this is a DEVS-based CCS, the experiments can be added during the simulation time. Both tests will begin with only one experiment at the beginning of the simulation. As the simulation goes, the user will suspend the simulation and then add a new experiment with a different configuration based on the previous configuration learning evolution. The user will

x_1	x_2	x_3	x_4	y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Table 4.1 – The truth table of the 4-Bit parity problem.

Experiment ID	Number of neurons
0	7
1	5
2	3

(a) Number of neurons inside the hidden layer.

Experiment ID	N	M
0	0.1	0.5
1	0.4	0.4
2	0.6	0.3
3	0.9	0.1

(b) Learning and momentum factors for learning.

Table 4.2 – ANN test configurations.

add an experiment or change its configuration via the implemented interface inside the CCS manager as shown in Figure 4.2. When the user chooses a specific experiment the configurable panel loads the current configuration and the user can change any of the learning factor (N), the momentum factor (M) or the activation function for both the *Hidden* and the *Output* layer atomic models.

Using the concurrent ANN configuration panel and the desired configuration in Table 4.2b, different experiments will be launched at runtime. First of all, the reference simulation (R) will get the experiment ID equal to 0, with a learning factor of 0.1 and a 0.5 momentum factor. As shown in Figure 4.3b the first curve represents the first configuration, at zone one, the simulation was suspended and the a new experiment was added using the ADD button in Figure 4.2, and then configured

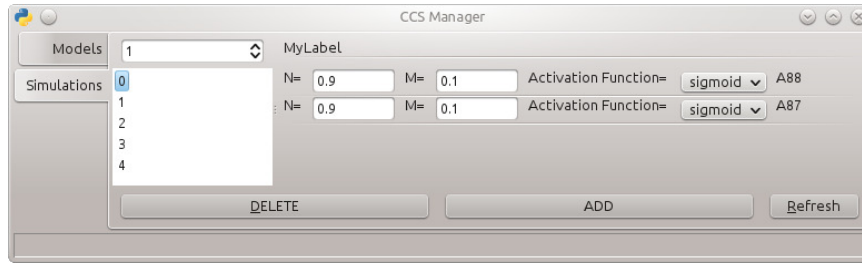


Figure 4.2 – The simulation panel interface of the ANN configurations.

with the 0.4 learning and momentum factor. With the new configuration the learning curve shows a faster learning rate, suspending the simulation two more times (points 2 and 3 in Figure 4.3b) to increase the learning factor and decrease the momentum factor. It is important to note that in order to create a new experiment at run-time, the ADD button makes a copy of the best current experiment and then applies the configuration changes - which is the reason of the curve splitting effect shown in the zones 1,2 and 3 of Figure 4.3b.

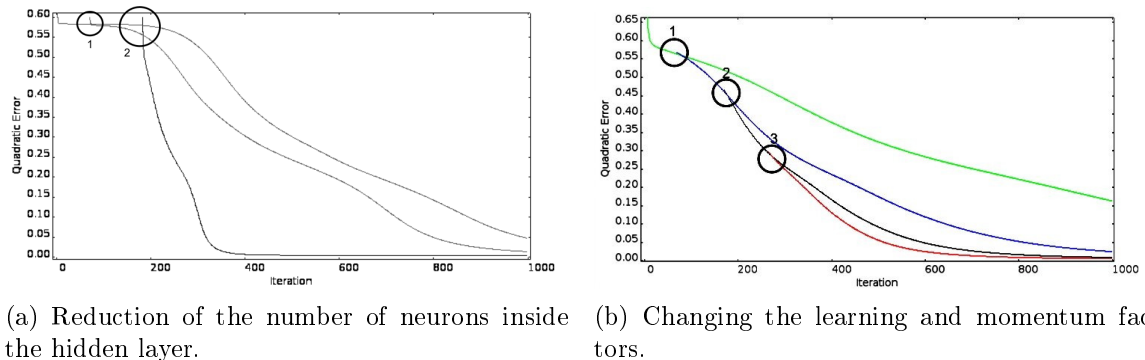


Figure 4.3 – ANN simulation results.

The second configuration will show the effect of changing the number of neurons inside the hidden layer without changing the learning, momentum factor or the activation functions. Figure 4.3a shows two splitting zones, 1 and 2, where the number of neurons goes from 7 in the reference simulation to 5 then to 3 respectively. The learning curves in this figure show a non smooth splitting in both zones 1 and 2. This is due to the sudden elimination of a neuron inside the hidden layer which introduces the error shown.

For further study of the ANN and the classification problem, some optimization can be placed in order to have a faster learning and an easier comparison between experiments and configurations.

4.3 ANN Modular Optimization for Classification

4.3.1 Introduction

Most of the used ANNs have a monolithic structure. A lot of the models work on the fully connected feed-forward ANN. This network performs well on the small input vector space dimension. The problem's complexity usually increases and the performance decreases with a growing input dimension. Another problem that faces the concurrent ANN is the simulation time. As mentioned previously, the overall simulation time is reduced by simulating multiple experiments at the same time. However, as the number of experiments increase the simulation time increases as well; which is an expected result. We still aim for the smallest simulation time and the easiest simulation configuration. For that reason, a deeper research and analysis on the ANN behavior and results took place.

The ANNs work best with a single task. Multiple ANN is a term used for separated ANN architectures. These separated ANNs are used if different information sources (different sensors) are available to give information on one object. Dividing the input vector into several smaller ones might not be a good idea. In order to increase the ANN classification a new multistage ANN is introduced. The difference between multiple ANNs and a multistage ANN is the hierarchical architecture and the input data form. To explain the idea of a multistage ANN, an input pattern analysis is presented followed by the multistage building process based on some analytical data.

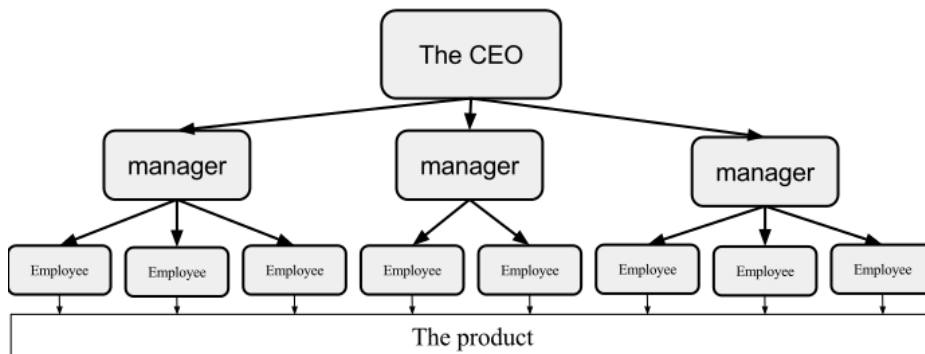


Figure 4.4 – The company job hierarchy.

The proposed multistage ANN will follow the organizations' and the companies'

job hierarchy as shown in Figure 4.4. In this hierarchy, the CEO (chief executive officer) receives all the ideas and thoughts then he distributes tasks to the managers [79]. Each manager has a smaller and more precise task. He must have all the information from the CEO in order to distribute the job to the employees. After that, each individual employee participates to produce a part of the final product. This final product is the result of the participation of every employee in the company, in other words, the participation of the three commanding levels of the company (stages). Every company has a different target and different products from one another, which makes it necessary for some analysis before the creation of the company's hierarchy. This makes the input pattern analysis the first task to create a multistage neural network.

4.3.2 Pattern Analysis

Each individual artificial neural network is considered as a brain or a processor where the work is distributed. The first stage is the responsible of the work distribution to the different brains (processors), in our case it is the ANNs. The work can only be well distributed if it was well understood and analyzed. The main goal is to identify and distinguish between the different data classes.

The data analysis at this stage consists of determining the similarity between the different classes (Example with classes: C1-C4) [80]. If two classes generate similar signatures or a very close behavior, then more processing power has to be dedicated in order to differentiate between them. An example with only four data classes is shown in Figure 4.5. If fault C1 and C2 are very similar, equally, C3 and C4, the identification process can be divided into two stages. The first stage classifies the four patterns into two groups (G1 and G2). One specialist ANN (A1) is needed to perform this task. As its only responsibility is to classify the data into these two groups, it can be more efficient and accurate to perform this task. The second stage consists of two different ANNs (A2,A3), one for each group. A2 and A3 are responsible for G1 and G2 respectively. A2 has to develop the criteria to distinguish between C1 and C2. Similarly, A3 has to develop the criteria to distinguish between C3 and C4.

As shown in the example, the number of groups (two in the example) has to be known to be able to create the stages and to know the number of ANNs needed. To

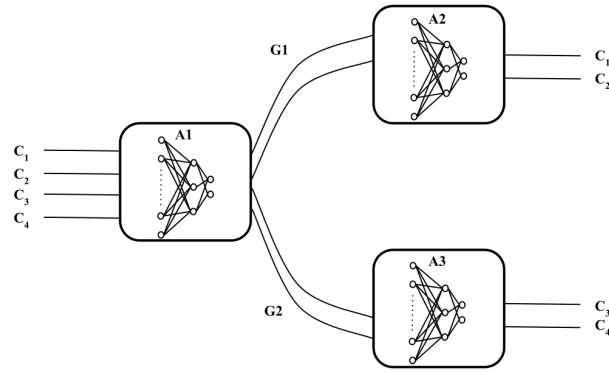


Figure 4.5 – Work distribution on a multistage ANN explanation example.

do so, a very simple statistical calculation is performed. This analysis is done only once to create the multistage architecture. For classification purposes, the Euclidean distance [80–82] calculation is chosen to help the architecture of a multistage ANN building.

The Euclidean distance between two points p and q is the length of the line segment connecting them (\overline{pq}). In Cartesian coordinates, if $p = (p_1, p_2, \dots, p_n)$ and $q = (q_1, q_2, \dots, q_n)$ are two point in Euclidean n -space, then the distance between them is given by: $\sqrt{\sum_{i=1}^n (q_i - p_i)^2}$.

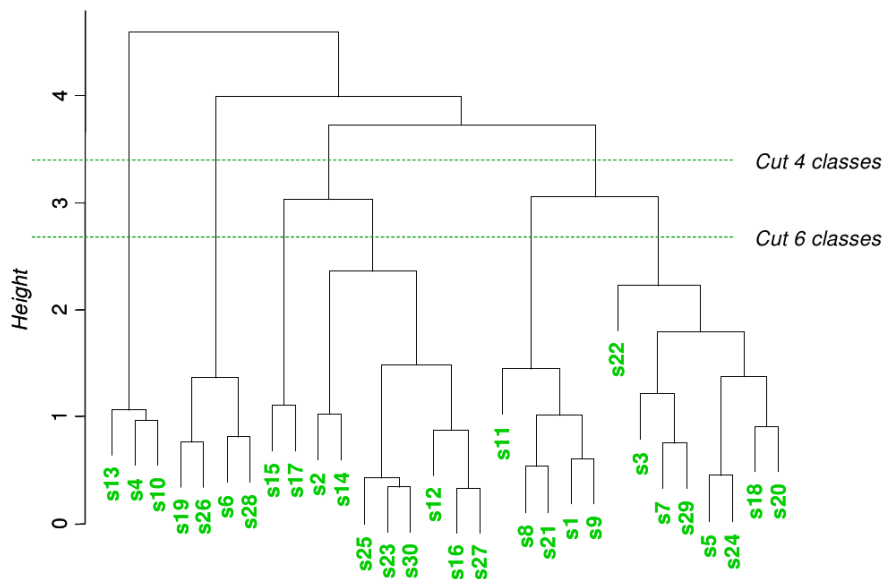


Figure 4.6 – Dendrogram cluster analysis on standardized Euclidean distance on random data.

A dendrogram (from Greek *dendro* "tree" and *gramma* "drawing") is a tree diagram representation used to illustrate the arrangement of a hierarchical clustering. Such a visual representation fits the needs of this thesis to group and categories the

given data to simplify an ANN work as it will be [53].

Figure 4.6 shows a random data to distinguish between 30 different input vectors (s1-s30). The graph shown in the figure has two different cut levels. The height of the higher cut makes it easier to differentiate between classes. Two cuts are shown in Figure 4.6, the first is a cut at the height of 3.3 showing four different classes, which means all the 30 input vectors can be easily classified into 4 classes. The next cut is at the height of 2.7 which shows 6 different classes. If two input vectors are not separated until the height of zero then they are exactly the same and it is extremely hard to differentiate between them.

The Euclidean distance is used as the metric distance between all of the 30 input vectors and is presented in the graph of Figure 4.6 in a tree form which we previously mentioned to be called the "Dendrogram".

4.3.3 Multistage ANN for Classification

The multistage artificial neural network was inspired by the enterprise hierarchy (Figure 4.4). The work distribution on a multistage ANNs is an interesting concept that comes as a solution for many complex problems [83–85]. A lot of the ANN research work uses a different unique concept different from all the others without a defined procedure that anyone could follow in order to increase the ANN performance all the times. The classification problems can be done using only statistical calculations [86], also widely used with machine learning and ANNs. Using both at the same time to enhance the overall performance is a further step towards a better system. For building a multistage ANN capable of recognizing more efficiently the input patterns the dendrogram is used.

The multistage architecture will be based on the cut applied in the dendrogram graph. As shown in Figure 4.6 at the height of 3.3 it is easy to distinguish between the four classes. By consequence, it will be easy for a neural network to perform the same task too. Then this makes the first ANN stage a differentiator between four classes - the role of a CEO in a company, where he gives different tasks to different managers. Then, at the second stage, four different ANNs take the job. Each ANN will be responsible for a simpler task, which is to identify between a maximum of 10 different input vectors. If at the second stage it is still difficult to distinguish between

all the input vectors, another stage can be added to make it easier to perform the classification task. Depending on the problem the number of stages can increase in order to perform the classification process.

4.3.4 Conclusion

This chapter sums the work done in three different working areas: the artificial neural network, the discrete event system specification formalism and the concurrent and comparative simulations. For every one of these areas a new concept was introduced and validated. First the DEVS formalism gained an optional concurrent behavior for all its atomic models with a simple implementation of the concurrent behavioral function without any change of the classic DEVS simulation kernel. The neural network was fragmented into different small DEVS atomic model, and then they gained a concurrent behavior in order to compare different simulation and lastly a classification optimization using statistical calculation is proposed - the multistage architecture. For the concurrent and comparative simulation the option of being able to add an experiment in run time was easily introduced with the integration of the DEVS formalism inside the DEVSImPy environment. At this stage of work, a concurrent neural network based on the DEVS formalism is created and optimized for the classification tasks using some statistical calculation to build a multistage ANN capable of solving complex tasks. It will be introduced in the next chapter to perform fault diagnosis in the WRIMs.

Chapter 5

Case of Study: Electrical Machine Diagnosis

5.1 Introduction

This thesis deals with multiple domains: discrete event system's modeling and simulation, artificial intelligence (ANNs), electrical machine fault diagnosis, and digital data compression. This chapter is the summary for work done in each domain. The work is inspired by the need of the wound rotor induction machine fault diagnosis of the direct usage of digital data through an artificial neural network.

A new DEVS-based feed-forward artificial neural network (FF-ANN) is presented in Chapter 2. A DEVS-based concurrent and comparative (**DEVS-based CCS**) simulation is presented in Chapter 3 in order to add new, concurrent and comparative features to the DEVS formalism without any changes in any DEVS simulator. This chapter brings out two application layers: (i) the concurrent DEVS-based ANN with a customized plug-in configuration interface to manage the concurrent experiments. (ii) The wound-rotor induction generator inter-turn short-circuit diagnosis using the new digital neural network.

This chapter deals with three major points:

- The pre/post processing data for neural networks and their effect on the network performance. A new digital transformation and compression for the digital input used to train and test the FF-ANN for wound-rotor induction generators.

- The usage of the DEVS-based ANN and applying a concurrent training, and showing a developed DEVSimPy CCS plug-in configuration panel to manage the concurrent experiments for the WRIM fault diagnosis.
- An efficient architecture of multistage neural network based on the Euclidean distance between the different training patterns and classes.

This chapter is organized into four sections. The first section deals with the ANN digital data pre-processing. The data pre-processing usually has a big impact on the neural network performance. This section applies new, efficient digital data compression for neural network classification needs. The data presented is composed of a pure periodic induction generator signal with different frequencies. The second section shows the use of the concurrent learning neural network and the simulation managing with the developed configuration panel. The third section shows the impact of the pre-processing technique on the neural network for the electrical motor fault diagnosis and a comparison with the uncompressed data. The last section brings an efficient study to create a multistage neural network capable of solving more complicated problems faster and efficiently. The multistage architecture is compared with the single stage ANN for machine diagnosis.

5.2 Short-Circuit Diagnosis

5.2.1 Set-up Description

The experimental set-up (Figure 5.1) has been designed in order to perform measurements on a lab scaled three-phase 5.5kW, 50Hz, 220/380V, 8-poles. A back-to-back voltage static inverter has been used to control the rotor currents. The WRIG is driven by a prime mover designed around a 7kW three-phase squirrel-cage induction machine and a 11kW programmable voltage static inverter (VSI). To emulate the wind speed and allow the system to operate in different modes and with different output power, the prime mover is controlled. Two operation modes are available: steady state (constant wind) and transient state (wind speed slopes). The back-to-back converter is able to control the DC bus voltage by the absorption of sinusoidal currents and imposing WRIG rotor currents with convenient magnitudes, frequencies and phased. A three normalized stator current taken at steady states as an example shown in Figure 5.2. The normalization was done as the part of the pre-processing steps. The data coding has been done with an accuracy of a 16-bit, which is the maximum available on the DAB.

This setup was designed to collect both stator and rotor currents of the WRIG by means of a data acquisition board with dedicated 16-bit analog to digital converters. The current sensors (basically AC current measurement) are based on magnetic cores with windings. The scaling factor for the current sensors is 0.1V/A and the frequency bandwidth is 1Hz-20MHz. Therefore, the output of the current sensor is analog. The digitizing process is performed with the data acquisition board having a signal conditioner at its inputs in order to scale the maximum magnitude and to pass through a low-pass filter in order to perform the anti-aliasing task. It is highly expected in future applications to use current sensors with a digitizer module inside each wireless transmission.

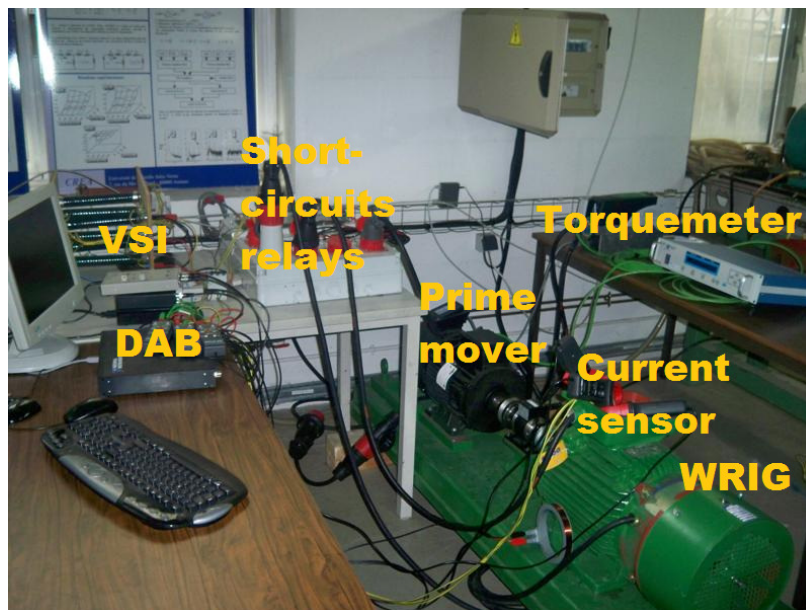
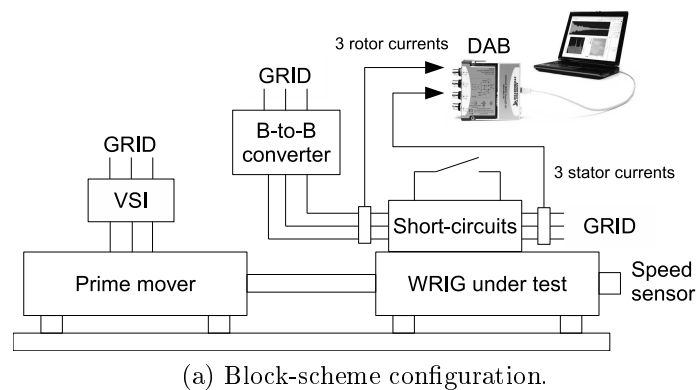


Figure 5.1 – Set-up for different tests.

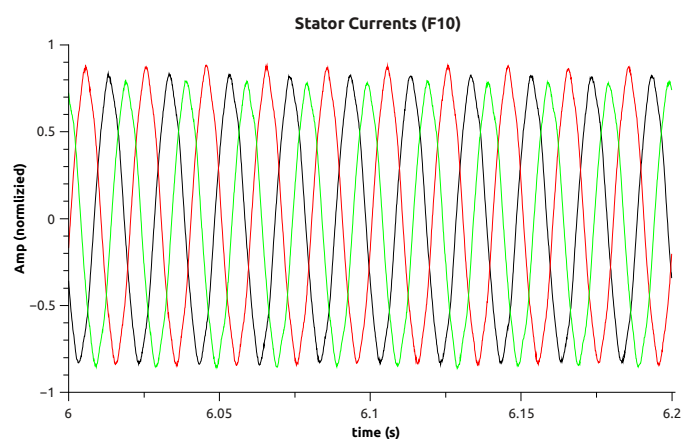


Figure 5.2 – Sample of 3 normalized stator currents (2000 samples each) at steady state with fault F10.

Short-circuited winding

The acquisition time has been set to 10s. The data is normalized, digitized and stored in files with a sampling period of 0.1ms. In order to perform a significant training for the neural network, 16 types of faults have been used depending on both rotor or stator sides and phases and coils. Fifty percent of the data has been used as training patterns for the neural network and the remaining fifty percent has been used for tests. At the current time, no transients have been yet considered.

The type of electrical fault used in all experiments is based on winding short-circuits (see Chapter 1.1.2). These faults have been tested with a rotor speed of 775rpm (rated load, slip=3.33%) and a synchronous rotating speed of 750rpm (grid frequency of 50Hz for 8 poles) at steady state.

Both stator and rotor windings have been modified by rewinding magnetic circuits on the first and last turn of each winding for the three phases. Therefore, there are two more windings per phase and per side (stator, rotor) on each phase. Stator and rotor windings have the geometrical configuration $(1 : N - 2 : 1)$ turns for a total number of N turns per phase (see Chapter 1.1.2). On the other end, every first and last turn of each phase and side has a switch in parallel to directly short circuit any of these turns. The total number of switches is 24. The direct short-circuit does not greatly affect the magnitude of the short-circuited current which never crossed over its rated value.

At this stage of work, a combination of short-circuits are considered as one fault **F**. In other words, multiple winding short-circuits are considered as one fault. For the time being, 16 different fault combinations are considered, 6 for the rotor side and 10 for the stator side. Each short-circuit introduces a different frequency in the currents which can be detected by the frequency analysis. Another trend is to detect those faults in real time without the expenses of the frequency analysis. To do so, some data analysis is required to be able to match needs and requirements for the detection mechanism.

5.2.2 Learning and Test Patterns Study

As previously discussed, data analysis is a very important task to improve the ANN performance. Two main data features are very important when it comes to data analysis: the nature (periodic, aperiodic, biased, unbiased, etc.) and the size (vector dimensions). The data used as the neural network input for the electrical fault detection (learning and validation) is consisted of: the values of 3 stator currents and 3 rotor currents when the WRIG is operating in both healthy and faulty modes at a steady state. For the electric current used in this application, the two points that we studied are the:

- Data type: all data patterns are composed of pure periodic. The rotor and stator operate at different frequencies. This difference in the operating frequencies has to be considered when it comes to data sampling time or data compression technique.
- Data dimension: Each pattern consists of 6 currents (3 rotor, 3 stator) for a periodic time of 0.2s. Each pattern is composed of 10 periods of the stator side (frequency of 50Hz) and 1/3 of a rotor period (frequency of 1.9Hz). Since the sampling period is 0.1ms, each pattern is composed of 2,000 points for every current. This means an input pattern for the neural network with a dimension of $2,000 \times 6$ (i.e. 12,000). Sixteen different faults have to be identified by the neural network as well as the healthy mode, making the input patterns consistent of $2000 \times 6 \times 17$ (i.e. 204,000). Each fault type is represented by ten patterns. The number ten has been chosen to ensure a good generalization while teaching the neural network each fault.

To be able to use periodic data as input to an artificial neural network, the data has to start with a fixed phase point. To be able to do so, a phase detector [87, 88] has to be integrated while collecting the currents to be an input to the ANN. Otherwise, the data has to be transformed or compressed in a manner to eliminate the phase influence on the neural network learning.

Another time domain analysis can be performed to predict how easy the classification can be. This analysis is based on the hierarchical ascendant classification (HAC), which is a method to represent and compute an index related to the distance between

the centers of gravity of each input vector to the neural network. Also another study is called the Euclidean distance (ED). These types of analysis have a tree presentation form called dendrogram. Dendrograms are useful to validate classification models [89]. They give a tree diagram that represents a hierarchy of input vectors based on the degree of similarity. This type of analysis will be used later in this chapter in order to build a multistage neural network.

As shown, the data type and dimension impose some difficulties for the neural network to learn the different electrical faults. Large periodic data might cause some learning difficulties. As it can be seen, our data has two difficulties: periodic and large dimension. With this in mind, data pre-processing techniques have to take place in order to transform this data to be fully optimized for the ANN learning.

5.3 Data Pre-Processing

Data analysis is one of the earliest pre-processing stages. The pre-processing is a process of inspecting, cleaning, transforming the data, in order to discover useful information. As it has been often evaluated, during learning and validation modes, the ANN exhibits poor performances when the input data are periodic with zero mean values. The same conclusion can be reached when the dimension of the input vector is too large. Usually, the complexity, redundancy and the large dimension of the input data used for training the ANN, leads to a weak performance and efficiency. Thus, most applications transforms the data into a new presentation before training the ANN. This data transformation is often application related and is still a very interesting research domain. In many cases, the data transformation (sometimes called pre-processing) techniques is one of the most significant factors to determine the ANN final performance [50,90]. In many applications, the dimensions reduction of the input vectors can be efficient to reduce the problem complexity and the learning time.

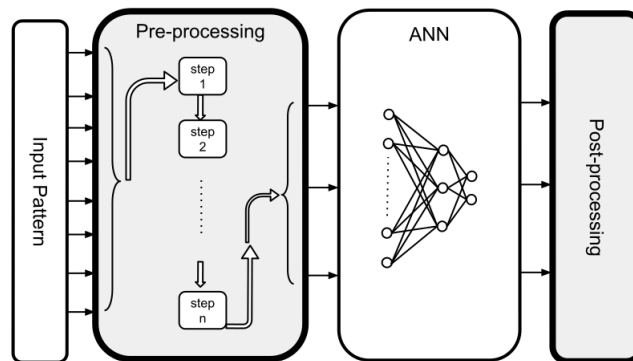


Figure 5.3 – Data pre/post processing for neural networks.

The ANN usually has data processing blocks that surrounds it (Figure 5.3). Both pre/post-processing are important steps to use the ANN efficiently. A new pre-processing technique for digital data is presented in this thesis, followed by a new vision of the post-processing by proposing a multistage neural network, which will be discussed later in Chapter 5.5. As shown in the figure, the pre-processing is a sequence of multiple steps. Some of these steps is considered as linear transformation, although later in this section a new digital compression technique is presented.

Linear Aspect

A simple linear rescaling of the input variable is the first step to be considered. The data rescaling is more required if different variables have typical values which differ significantly. By applying a linear transformation, all input are in the same scale. To do so, each input variable is treated independently. For each input x_i we calculate its normalized value $\bar{x}_{i,0-1}$ using:

$$\bar{x}_{i,0\ to\ 1} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

where $\bar{x}_{i,0\ to\ 1}$ is the data point x_i normalized between 0 and 1. For a more centralized and normalized data set, with the zero being the central point, the following equation is used instead:

$$\bar{x}_{i,-1\ to\ 1} = \frac{x_i - \left(\frac{x_{max} + x_{min}}{2}\right)}{\frac{x_{max} - x_{min}}{2}}$$

where x_i represents the data point i , x_{min} is the minima among all the data points, x_{max} is the maxima among all the data points.

For the WRIM data, the currents' values oscillate between [23A,-23A]. For the absolute values, the x_{max} is equal to 23 and the x_{min} is equal to 0. The linear pre-processing scales the data in such way that all neural network's weights remain small and avoid the saturation of the activation function. On the other hand, this data transformation is not the only pre-processing step. In some application domains, the data compression is recommended to reduce the dimension of the input vectors and reduce the learning time. The next subsection shows a new digital compression technique for periodic scaled data.

Digital Compression Aspect

The electric currents are periodic, which means the phasing has to be considered, otherwise it has to be eliminated. One stator current with a 2,000 acquisition points will be used as an example (Figure 5.4) to show the digital compression steps.

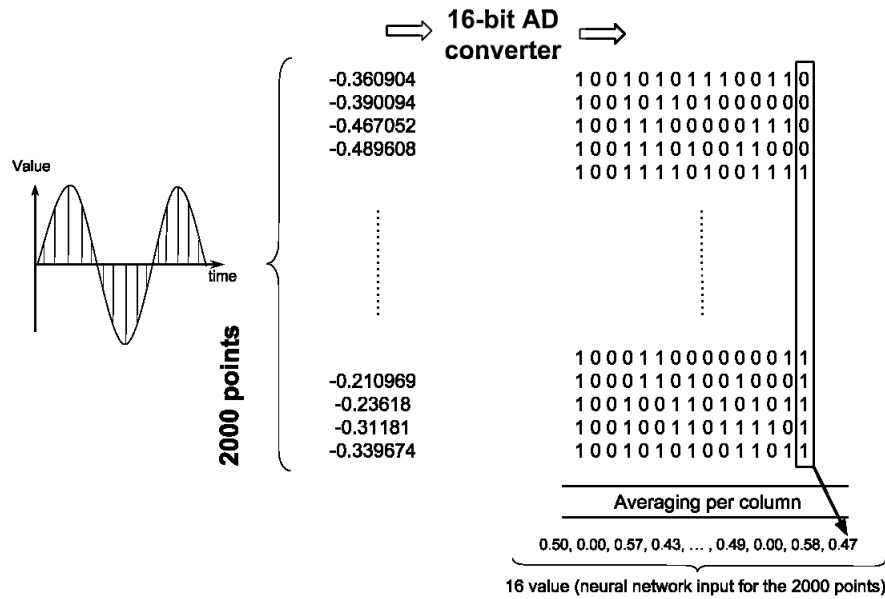


Figure 5.4 – Digital compression procedure.

The new digital compression technique that will be applied on the electric current to reduce the input dimension space and eliminate the phase influence needs two steps:

- Digitizing the data: Nowadays, it is common to use digital sensors. On the other hand for an analog sensor, a 16-bit AD (analog to digital) converter is used to convert the analog data. Figure 5.4 shows a 2000 points vector that needs to be converted and compressed. First, each signed float point is converted into a train of 16 bits. This step transforms the 2000 float points into a matrix of 2000×16 . This matrix contains 2000 rows of ones and zeros. Any digital data is presented with digital bits, is considered to be composed of MSBs (most significant bits) and LSBs (least significant bits).
- Averaging binary columns: This process converts the matrix of 2000×16 to only one row of 16 values (Figure 5.4). As the average has been calculated, the float points order is no longer important, which means the signal phase is not available any more.

These calculations are neither time nor frequency transformations but a new binary technique which is very simple in term of operation. As the binary data is a combination of ones and zeros, the averaging represents the percentage of ones per column.

As it is a 16-bit analog to digital conversion, we estimate that MSBs might contain more information than the LSBs. This estimation is based on the constant change of the LSBs, while the MSBs are represent more of the currents envelope where more information exists. This concept has been studied in [91] and all validations and comparisons are shown in the next section of this chapter.

The two digital compression steps, digitizing and averaging, are the only steps used to compress the data. All kinds of compression methods must certainly introduce some data loss. However, if too much information is lost during pre-processing it will obviously lead to performance deterioration. In the digital compression case, the redundancy of the periodic data has been eliminated, and the input dimension is reduced by a very large factor making the ANN learning much faster and easier.

The compression ratio (CR) for this technique is inversely proportional to the number of bits used by the AD converter ($B = 16$). This can be represented by: $CR = \frac{N}{B} = \frac{2000}{16} = 125$.

The input of the ANN for the machine diagnosis is composed of 6 electric currents, each consisting of a 2000 original values and will be compressed into a 16 values after the digital conversion and the averaging. This transforms implies a new neural network input composed of a 16*6 input values making each input vector composed of **96** values rather than 12,000.

Summary

In this section two categories of pre-processing were presented: linear and digital compression aspects. The linear aspect was already well known and used with different application domains. It was applied on the periodic stator and rotor currents in order to normalize the currents and insure that they will be on the same scale. The data scaling is always important to equalize the different impact of data types. The presented digital compression is a column averaging applied on a 16-bit converted data to produce a percentage of the ones per column. This eliminates the data periodic redundancy. The digital averaging reduces the input vector dimension with a CR equals to $\frac{N}{B}$ where N is the number of inputs and B is the number of bits used for the conversion. Both aspects are going to be validated with a comparative and concurrent DEVS-based ANN implemented inside the DEVSimPy environment using

the 6 electric currents of the WRIM, to localize the short-circuited windings.

5.4 ANN Configuration and Simulation

In this sub-section, 12 electrical faults are used concurrently, the first 6 faults are on the stator side and the other 6 are on the rotor side (F1-F12). On the first step, a single stage ANN is used to differentiate between the 12 different faults. As the number of bits used by the AD converter has a significant impact on the neural network performance, different tests are made. First, a full 16 bits study is made then followed by a 12, 9 and 6 bits variant. It was previously mentioned that the MSBs might contain more information than the LSBs, for that reason the 12, 9, and 6 bits variant are all used from the MSBs side.

16 bits Patterns

As the DEVS-based CCS is used, different configurations are tested and compared concurrently. Table 5.1 represents the different configuration combinations for the ANN. It is recommended that the learning factor (N) and the momentum factor (M) oscillate between 0 and 1. The optional transfer functions are: linear, hyperbolic tangent, and sigmoid. Only the sigmoid and the hyperbolic tangent were used, they are the most common and reliable activation functions for non-linear problems. For all experiments of this sub-section, 4 outputs for the ANN are considered. The outputs are represented with 4 bits; the ANN can differentiate between up to 16 different classes with these 4 bits. In our case we need only 13, 12 for the faulty patterns and one represents the safe mode.

Experiment ID	N	M	Activation function	Neurons (Hidden layer)
1	0.1	0.1	tanh	20
2	0.9	0.1	tanh	20
3	0.9	0.1	sigmoid	20
4	0.9	0.1	sigmoid	50
5	0.9	0.1	sigmoid	70
6	0.9	0.1	sigmoid	60
7	0.9	0.5	sigmoid	55
8	0.9	0.2	sigmoid	54
9	0.7	0.1	sigmoid	56
10	0.9	0.1	sigmoid	56
11	0.9	0.5	sigmoid	56

Table 5.1 – ANN configurations.

All experiments are simulated concurrently using the concurrent DEVS-based ANN. Different number of neurons inside the hidden layer is tested with different configurations (Table 5.1). The "*Errorgenerator*" atomic model inherits the oscilloscope display to show the error evolution curve during the learning phase as shown in Figure 5.5.

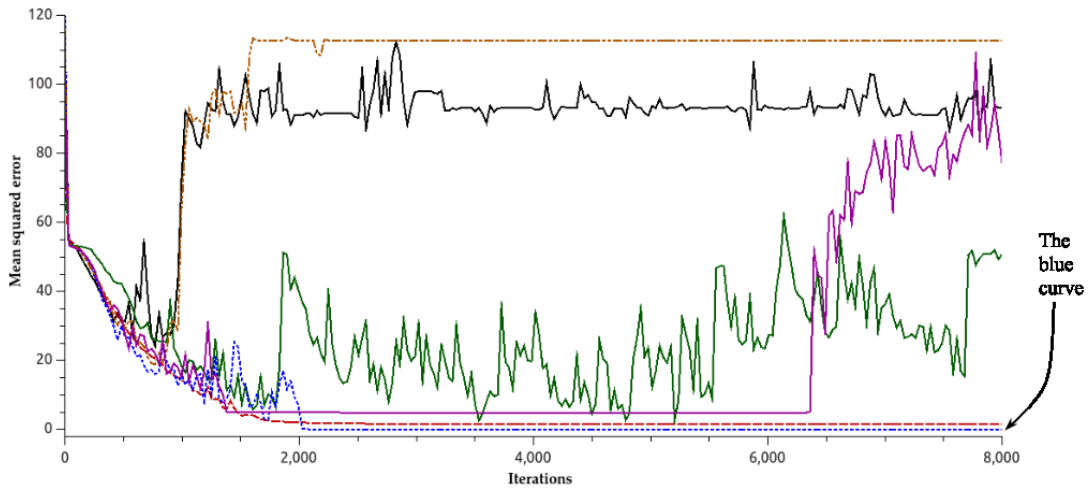


Figure 5.5 – Mean squared learning error for the concurrent simulation presented in Table 5.1.

Figure 5.5 shows 6 different learning curves among the eleven configurations presented in Table 5.1 in order to be visible. The presented curves show the mean squared learning error over 8000 iterations. As it can be noticed from the figure, the error curves converge only when a suitable configuration is adapted and diverge when inappropriate configurations exist. As shown, the lowest learning error comes from the experiment ID number 9 (blue curve in Figure 5.5) where the learning factor is 0.7, momentum factor is 0.1, activation function is sigmoid and 56 neurons inside the hidden layer. This configuration confirms some of the recommendations made by [23, 50] as the number of hidden neurons is equal to the number of inputs.

The best result among the 11 concurrent simulations is detailed in the 4 columns Table 5.2. The first column shows the fault ID number (detailed in previously in Chapter 1.1.2), the second column represents the desired/optimum output for the neural network. The third gives an output example for the real output of the ANN during the test. The last column gives an error percentage per fault during the tests.

Each row represents the data for a specific fault. For example, the F7, the network

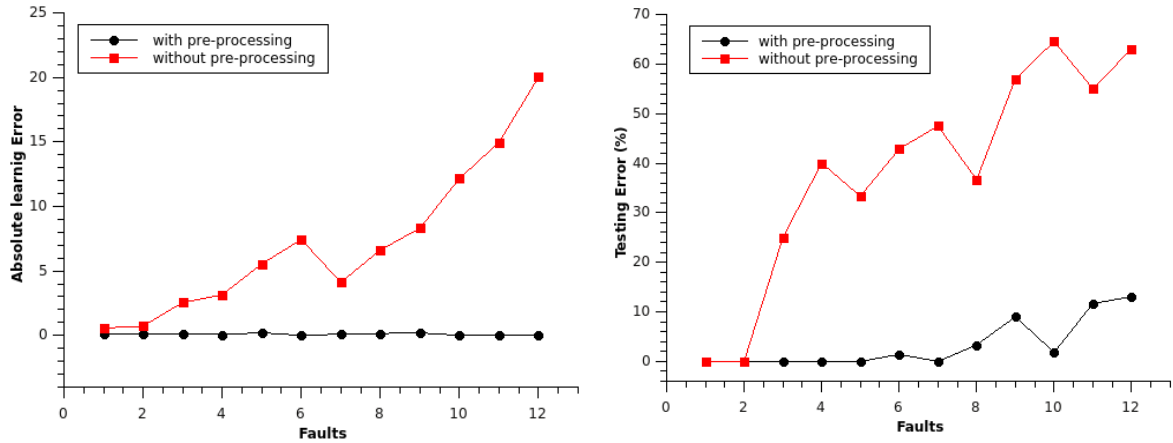
Fault type	Desired output	Real output	Test error [%]
Healthy	0, 0, 0, 0	0.08, 0.05, 0.03, 0.05	0.0
F1	0, 0, 0, 1	0.07, 0.09, 0.04, 0.90	0.0
F2	0, 0, 1, 0	0.07, 0.05, 0.98, 0.04	0.0
F3	0, 0, 1, 1	0.05, 0.15, 0.91, 0.93	0.0
F4	0, 1, 0, 0	0.09, 0.93, 0.10, 0.08	20
F5	0, 1, 0, 1	0.06, 0.80, 0.26, 0.90	40
F6	0, 1, 1, 0	0.10, 0.92, 0.88, 0.07	0.0
F7	0, 1, 1, 1	0.41, 0.71, 0.86, 0.77	70
F8	1, 0, 0, 0	0.99, 0.05, 0.04, 0.01	0.0
F9	1, 0, 0, 1	0.90, 0.02, 0.18, 0.75	0.0
F10	1, 0, 1, 0	0.92, 0.08, 0.94, 0.04	0.0
F11	1, 0, 1, 1	0.95, 0.08, 0.85, 0.91	0.0
F12	1, 1, 0, 0	0.90, 0.98, 0.09, 0.19	0.0

Table 5.2 – Results for classification of the first 12 faults with a 16-bit AD (simulation ID = 9).

should generate an output (desired output) of $[0, 1, 1, 1]$ from the 4 output neurons. The experiment was not able to correctly detect more than 30% of the test patterns (three out of the ten tested). An example of the real output generated by the network is $[0.41, 0.71, 0.86, 0.77]$ for the same F7.

Using all of the 16-bit values for each electric current, the single input pattern to the neural network is composed of a 96 input values (16 values per current, for 6 currents). To differentiate between all of the 12 faults and the safe mode 4 output neurons are used. The performance of the ANN with the digitally compressed data shows a better learning convergence over the non compressed data (Figure 5.6). The comparison is made with twelve different simulations. The neural network is first tested to classify only one fault (F1) with the WRIM safe mode. At this stage, both compressed and uncompressed networks performed well. Second, an additional fault (F2) was added to the classification, then the fault F3 was added and so on, until the last simulation was made with all the 12 faults plus the safe mode. In Figure 5.6a, the quadratic error for the learning shows the benefit of the digitally compressed periodic data.

In summary, the data compression shows a much better performance to classify and differentiate between the twelve faults and the safe mode. The compression with a 16-bit AD converter transforms and compresses the input patterns from a 1200 input values vector to only a 96 values with a CR of 125. After ten faults the



(a) Quadratic error for the 12 faults classification comparison. (b) Test error percentage for the 12 faults classification comparison.

Figure 5.6 – Learning and test error comparison between compressed and non-compressed data.

ANN performance decreases. One step further is to test different AD converters with different number of bits.

Variant Bit Patterns

After validating the digital pre-processing technique, it is very interesting to see the impact of changing the number of digits used to code the analog data extraction from the test bench. All results presented above are based on 16-bit AD converter and used an accuracy level of 4 digits after the decimal point.

Most commonly the right hand side values (LSBs) of the input patterns remains unchanged. This is due to the constant change of the LSB of the converted data, making the averaging are always around to 0.5. On the other hand, the MSBs of the converted data changes are less frequent which give different average values. This kind of unchanged input values introduces data redundancy which is unhealthy to train the neural network.

Finding the best accuracy needed by the ANN to be able to distinguish between the different faults is the goal of this sub-section. Three different set-ups are made to get different levels of accuracy using:

1. Three digits after the decimal point with a 12-bit AD converter.
2. Two digits after the decimal point with a 9-bit AD converter.

3. Only one digit after the decimal point with a 6-bit AD converter.

As the AD converter influence the number of input data, Table 5.3 shows the different number of input corresponding to the different AD accuracies. The number of inputs is a function of the electrical currents' number and the number of bits used by the AD converter. For a 12-bit AD converter with 6 currents, it gives $16 * 6 = 96$ ANN inputs. To make a fair comparison, the NB has been changed keeping all other ANN parameters unchanged (based on the best results with the 16-bit AD converter: one hidden layer, 56 hidden neurons, sigmoid transfer function and 4 output neurons). As

NB of bits	ANN inputs	Training quadratic error	Testing error [%]	Iterations
6	36	0.12	5.3	10,000
9	54	0.15	10.0	10,000
12	72	0.25	13.9	10,000
16	96	0.25	13.9	12,000

Table 5.3 – Influence of the AD converter accuracy.

shown (Table 5.3) the ANN performances are improved when the number of bits used by the AD converter in the pre-processing mode is reduced (only using the MSBs). This means that the ANN can find the fault signature inside only the average of the MSBs and additional data is considered as noise. The 6-bit AD converter is the minimal configuration that can be used to encode the different currents with a minimal accuracy of only one digit after the decimal point. The LSBs average does not offer to the ANN informative data because of the always close to 0.5 value due to the constant change of these bits.

To resume the different simulations, the one digit after the decimal point with a 6-bit AD converter proved to be the best simulation result with a 0.12 global quadratic error value and a 5.3% test unrecognized patterns. The error is meanly concentrated in F7 and F5, which their digital signature might be very similar to other faults making the differentiation between them very difficult to the ANN. These results are for the 12 faults and the safe mode operation with ten patterns for each mode. The same amount of patterns is used for testing.

Summary

The new digital compression technique has been presented in order to reduce the input dimension space. It also eliminates the phase influences of the periodic data. The technique was applied to twelve different electrical fault of the WRIM (F1 - F12) with a significant improvement of the neural network performance over the uncompressed data.

As a matter of fact, a very large number of unique faults \mathbf{F} are tested with this technique as we sometimes consider multiple short-circuits as one fault (i.e. F1 is a short-circuit on the first two turns of the first two phases and on the first turn of the third phase of the rotor side). The number of the combined faults can be very large. An artificial neural network even with the best pre-processing techniques might have difficulties to classify and recognize all existing faults. As a result, this problematic tends to limit the fault detection capabilities with the ANNs. To solve this problematic, a multistage ANN is proposed to take over and maximize the number of faults that can be detected.

5.5 Multistage Optimization

The idea of multistage neural network is inspired by several researchers, but also by the multi-core processing solutions. The idea is based on distributing the calculations and the problems on several processing units. Each processing units has a smaller task to do. As we talk about neurons and neural networks, humans tends to work together to get jobs done faster and more efficient.

The idea of a multistage neural network is presented by several researchers [81–85]. To be able to understand how the multistage neural networks might work, a comparison with human behavior can be applied. Humans tend to work together to be able to get work done faster and more efficient. Each person handles one specific task, that makes it easier to understand and faster to accomplish. Correspondingly, the multi-core processing units tends to solve problems of the single processor (heat, complexity, etc). Each processing core is able to work independently from the other cores, the results are collected and assembled together in order to have the final result. In like manner, ANN has to work for difficult tasks, distribute the work and collect the

final result. In order to do so with ANNs, two steps are necessary to make this idea to work. First is to analyze the input patterns and second is building the multistage architecture.

5.5.1 Patterns Analysis

For the statistical calculations and presentations, the R-project for statistical computing [92] is used on a 64-bit Windows 7 Professional machine with an Intel® Core i5 CPU M460 Processor (4M Cache, 2.53 GHz) with 4GB of RAM. The data used for analysis is composed of a two input vectors per fault for all of the 16 faults (F1 - F16), making a total of 32 input vectors. Each vector is composed of the 96 values (for the 16-bit accuracy level, explained in the previous subsection). For the multistage purposes, the dendrogram will visualize the similarity between the different patterns. Figure 5.7 shows the similarity index between the 32 input vectors (Euclidean 96 di-

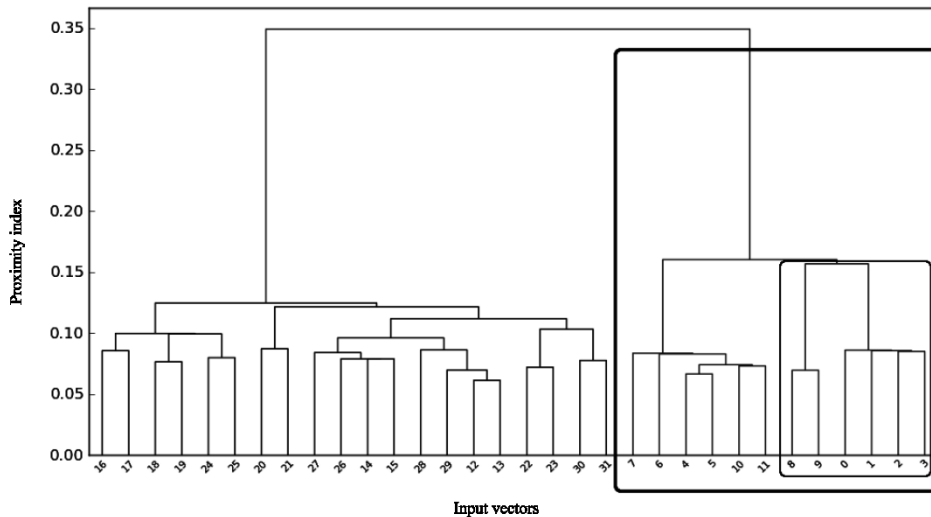


Figure 5.7 – Dendrogram for similarity study between 32 input vectors.

mensional space). The X axes is composed of a 32 values (from 0 to 31) representing the 32 input vectors, each fault got two vectors. F1 is represented by vector 0 and 1, F2 with 2 and 3, ... , F16 with 30 and 31. The Y axes represents the similarity index calculated by the R-project for statistical computing using the Euclidean distance between the different vectors. High index means less similarity, as the index goes higher the classification gets easier.

In the Figure 5.7 two main branches are created separating F1, F2, ..., F6 (rotor side faults, pattern from 0 to 11) from F7, F8, ... , F16 (stator side faults, patterns

from 12 to 31). The first thing to notice is the difference between the rotor side fault patterns and the stator side fault patterns. With a closer look on the rotor side, another two main branches are created separating F1, F2 and F5 from F3, F4 and F6. Even closer, F1 and F2 are still separated from F5. This analysis can be inspiring to create three different levels of classification. The first is to find out the difference between faulty stator, faulty rotor or the healthy mode. The second level is to find which phase is short-circuited, and the third stage determines on which turn the short-circuit occurs. Based on the dendrogram (Figure 5.7), three different levels are detected, the multilayer ANN architecture is built in the next subsection based on this analysis.

5.5.2 Multistage Architecture

The WRIM 16 fault patterns are analyzed previously with a statistical analysis given by the previous dendrogram in Figure 5.7. The goal is to distribute equally the classification difficulties on all of the ANNs used in the architecture. Figure 5.8 unveils

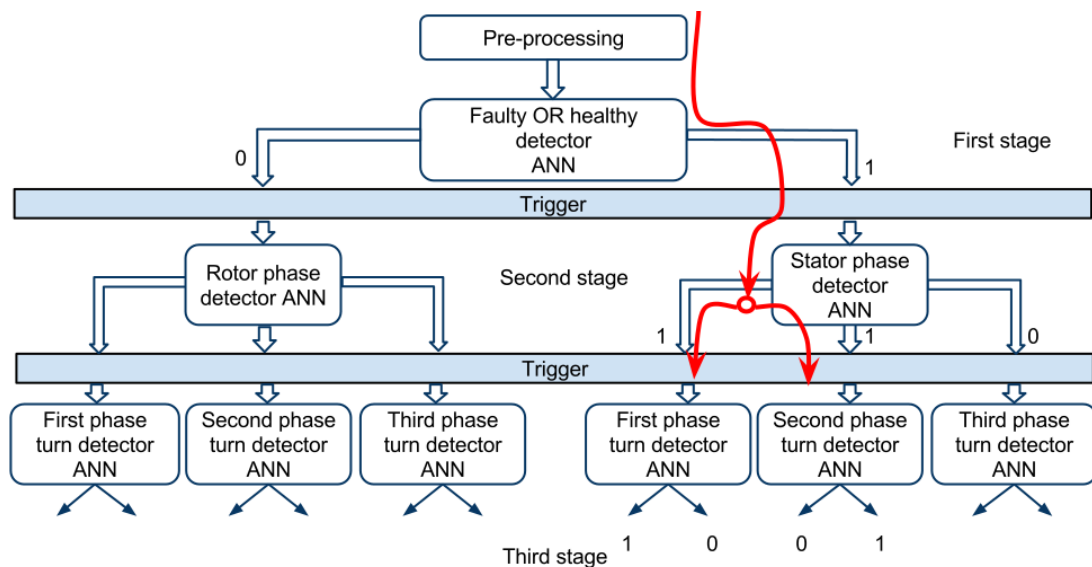


Figure 5.8 – Proposed multistage neural network for WRIG short circuit classification.

the proposed triply staged ANN composed of nine ANNs distributed as follows: (i) one in the first stage responsible to distinguish healthy, rotor side fault and stator side fault. (ii) two in the second stage, one for each side (rotor, stator) to detect on which phase the short-circuit took place. Both are also capable to detect multiple short circuits on different phases at once (shown in the example later). (iii) six ANNs

in the last and third stage, one for each phase (three-phase) on both sides (rotor and stator).

This proposed design came from the similarity index study, as previously seen in Figure 5.7. The dendrogram clearly is divided into two branches, which the reason for the first stage role has been chosen. The second and the third stages are chosen as phase identifier and turn identifier respectively. The patterns tend to be more similar when the short circuit on same phase and particularly on the same turn are common.

For all three stage ANNs has the same input provided with the 96 values input vectors representing the compressed data of the electrical machine with the 6 currents with using the 16-bit AD values and an accuracy level of 4 digits after the decimal point. All ANNs also are configured with the back-propagation configuration of a learning factor = 0.7, a momentum factor= 0.1, 56 neurons in the hidden layer, a bias = 1 and a sigmoid activation function.

Stage one: has two output values that can be either ones or zeros. An output of "00" means a healthy mode, "10" means a rotor side fault and "01" means a stator side fault.

Stage two: Both sides ANNs have three output values, one for each phase. As this stage is only triggered when there is short-circuit on this side, the output cannot be "000". When short-circuit is detected in a phase, the associated output bit shows "1". When three short circuits, one in each phase are present, the ANN output would be "111". When the output is "1" it triggers the associated ANN in the next stage to localize the short-circuit turn.

Stage three: Both sides ANNs have only two outputs, one for each turn. The same token of the previous layers for the output numbering is applied. The final output from the different ANNs will represent the short-circuit localization as shown in the example below.

Example¹ (Figure 5.8): F14 will be taken as an example, to show how would each stage works in order to get the final result which is to identify all of the 17 fault as well as the healthy mode. F14 is short-circuit on the first turn of the first phase and the second turn of the second phase. The 96 values of this pattern enters the first stage in the only ANN present there. The output from the first stage is: "01"

¹In this example real simulation outputs are shown.

triggering only the stator phase detector ANN in the second stage. In its turn, the second stage ANN receives the 96 input values and generates an output of "110". This second stage output, triggers two ANNs in the third (last) stage, which are the first and second phases turn detectors ANNs. Both triggered ANNs in receive the 96 input pattern and generates outputs. The first ANN generates an output of "10" which indicates the existence of short-circuit in the first turn of the first phase in the stator side. The second ANN generates an output of "01" which indicates the existence of short-circuit in the second turn of the second phase in the stator side. Combining the results of the third stage confirms the short-circuit on the first turn of the first phase and the second turn of the second phase which represents F14.

Simulation Results

The results presented in this subsection represents the best results obtained by the multistage ANN, pre-processing techniques to compress periodic data and configuration comparison using DEVS-based CCS for Feed Forward ANNs. Seventeen different simulations are executed to get the results presented in Figure 5.9. The first simula-

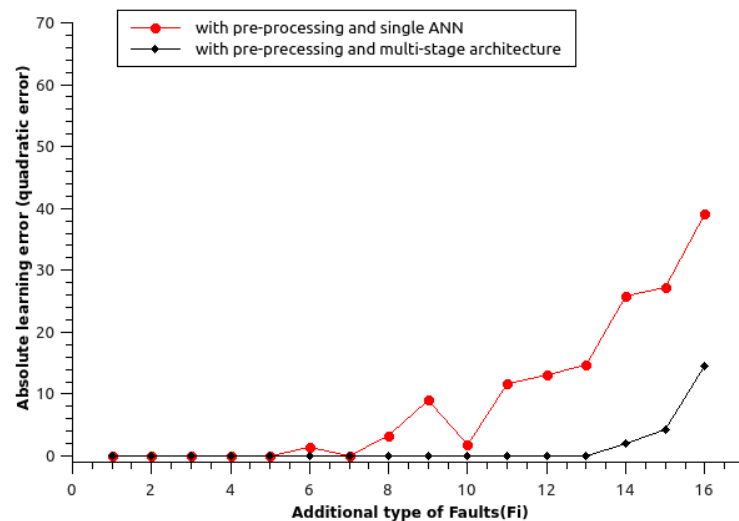


Figure 5.9 – Absolute error, comparing the performance of a single ANN to the multistage architecture.

tion is a classification between healthy mode and the fault F1. For each simulation after that another fault was added, until the 17th simulation represents the ability to classify all of the seventeen faults and the healthy mode. In the figure a comparison

between a single ANN classification capability and the proposed multistage ANN.

The single ANN gets an acceptable error rate with only 12 electrical faults, with more faults to classify, the error goes higher until it reaches a 38% test error rate with the 17 input vectors. On the other hand the multistage ANN is capable to get only test error rate with the same number of faults. The performance goes to the multistage architecture with a 28% better classification rate.

Added fault ID	Single ANN	Multistage ANN
F1	52	12
F2	547	295
F3	840	820
F4	3283	894
F5	1039	1103
F6	>20000	1429
F7	12696	1570
F8	>20000	7573
F9	>27000	7791
F10	>20000	7711
F11	>20000	6230
F12	>20000	5434
F13	>20000	7531
F14	>20000	>19000
F15	>20000	>15000
F16	>20000	>20000

Table 5.4 – Iteration number needed to get the best results during the learning mode.

Table 5.4 presents the number of iterations needed to get the best results for both with/without the multistage architecture for each of the seventeen simulations. The number of iterations for the multistage ANN represents the sum of all iterations needed to train all of the nine used ANN in the architecture. As the numbers tell from the table, the multistage needs much less iterations to get better results than a single ANN. This is phenomenon is the result of the problem distribution on multiple ANNs.

5.6 Conclusion

The fifth chapter might be the meeting point between the theoretical concepts and the application domains. The aim of this chapter is to get the best out of the ANNs using the CCS DEVS-based simulation and apply it for machine diagnosis and pattern classification.

This chapter presents a set-up description and the acquisition equipments used to generate the training and test patterns. It also presents some pattern analysis and studies made to verify the data quality and determine the type of pre-processing needed for ANNs. It has been noticed that the data is significantly large and periodic. Both criteria are not advisable for ANN uses the pre-processing compression technique based on digital data was introduced. The proposed compression technique has a CR of 125 using a 16-bit AD converter using multiple precision levels of 4 digits after the decimal point or less. In order to test the new technique, a CCS DEVS-based ANN was build and multiple configurations are tested concurrently. All tests were developed inside the DEVSImPy environment. The ANN performance has increased 50% using the proposed pre-processing technique based on the digits data for a 12 unique faults.

The final stage in the ANN multistage architecture approach is based on the input vectors analysis. The study is based on the Euclidean distance between the input vectors creating a similarity index between the different input patterns. This index is presented in a form of a dendrogram (a tree representation). The calculated index gives the idea of how easily the classification can be performed, sometimes multiple ANNs are needed to solve the problem. For WRIG diagnosis, nine different ANNs were used in a three level architecture. The proposed design improved the overall performance enabling up to 16 different electrical faults detection and localization. These results are about 30% more efficient than a single layer ANN.

General Conclusion and Perspectives

DEVS is a hierarchical formalism description for a discrete-event system. The DEVS formalism differentiates between modeling and simulation, the simulation is done automatically by a generic simulator. The DEVS modularity can be very beneficial to an artificial intelligence branch known as the artificial neural network (ANN). The ANN is known for its black box behavior, as it receives input and generates output depending on the training it received. It was commonly introduced in the DEVS environment as one configurable atomic model. To change the learning algorithm or the network architecture, the code inside the model is modified. A solution to eliminate this black-box effect and make the ANN more flexible and easy to develop is introduced in chapter three. This solution consists of fragmenting the feed-forward neural network into a DEVS library composed of atomic models easy to modify and replace.

The DEVS-based ANN has two groups of models: feed-forward calculations and the learning models. This modeling concept ensures the separation between the feed-forward calculations and the learning algorithms. This separation makes a new tool for mathematicians to easily replace and test new learning algorithms or different architectures. The new ANN modeling is composed of four different atomic models: calculation (hidden and output layers), non-calculation (input layer), Error-Generator and Delta-Weight models. The calculation and the non-calculation models are the location where feed-forward calculations takes place, and the Error-Generator with the Delta-Weight are where the learning algorithms are implemented. This ANN's DEVS library is implemented inside the DEVSSimPy environment offering a graphical user interface for the DEVS modeling and simulation. The DEVSSimPy environment also

offers the ability to add plug-ins on top of the atomic models in order to add some functionalities. The Error-Generator has a plug-in to show the learning or the testing error curves, making it easy for the user to visualize the results. The DEVS-based ANN can also benefit from the DEVS extensions.

As DEVS becomes more popular, researchers find the need to add and develop some enhancements to the formalism. Adding new behavior to the DEVS formalism is called a DEVS-extension. Sometimes, adding extensions to DEVS requires the simulator modification. The BFS-DEVS is an extension to DEVS that adds a behavioral fault simulation. This extension is used for fault simulation in digital circuits with a concurrent behavior. At the same time, changing the simulator may lead to an incompatibility between extensions, which can reduce their usability.

The concurrent and comparative simulation (CCS) is a concept developed in order to compare different simulations, reduce the overall simulation time and the manual configuration work. To add this interesting concept to the DEVS simulator, that might need to create a DEVS-extension and modify the classic DEVS simulator. A new generic solution inspired by the BFS-DEVS extension was introduced in this thesis to extend the DEVS classic behavior with an additional concurrent behavior called the DEVS-based CCS.

The DEVS-based CCS, is a concurrent DEVS simulation based on runtime code modification. The main idea of this generic solution is to modify the atomic model in the manner to keep the interface required by classic DEVS simulator. This idea makes any concurrent atomic model able to be simulated with any generic DEVS simulator. The DEVS-based CCS is implemented in two parts. First, an additional concurrent behavioral function is added to the atomic model along side with an additional object called the model signature. The second part is composed of a DEVSImPy plug-in, always executed before the simulation starts, and link the concurrent behavioral function with a classic DEVS transition function. By default, the concurrent behavioral function is linked to the external transition function. All signatures are managed by a signature manager, which is responsible of adding or deleting any experiment during the simulation time. For simulation managing, the signature manager has a customizable interface that can be adapted to any type of applications. The ANN can be a very interesting application to apply the concurrent DEVS extension.

The DEVS-based CCS is combined with the DEVS-based ANN to give a new concurrent ANN able to simulate different configurations at the same time. This concurrent ANN is used to find the best configuration possible for the electrical machine diagnosis.

The number of wind farm is growing every year, as a result condition monitoring is an essential tool for wind turbine generators to reduce the maintenance cost and the machines reliability. It is well known that any AC electrical machine can still operates with shorted turns even at its rated load. It is clear that early stage detection can prevent a full winding failure. The condition monitoring can be performed by several methods, but the electric monitoring is the most common among them. This thesis an ANN with back-propagation learning algorithm has been used to be trained directly by signals coming from sensors eliminating the analysis mode and simplify the architecture for diagnosis. The ANN used is based on the concurrent DEVS-based ANN developed also in this work making the different ANN configuration comparison a lot easier and interactive. However, periodic or very large input vector (hundreds of inputs) can result a poor performance for the ANN. Therefore, a new compression technique has been presented in this thesis in order to eliminate the periodic effect and reduce the input vector dimension. The compression technique is based on digitizing and averaging data using the benefit of the most and least significant bit of the digital data. The compression technique used with the DEVS-based ANN has shown more than 30% increase in the overall performance trying to localize twelve different short-circuited turns on both stator and rotor sides.

Last but not least, the multistage architecture for electric fault diagnosis is a new study for designing a multistage ANN based on statistical analysis. For sure the digital compression technique proved performance increase over the non-compressed data, but still does not give the expected results when seventeen different faults are tested. Therefore, this thesis presented a statistical analysis based on the Euclidean distance between the input vectors creating a tree graph also known as the Dendrogram. Based on this graph, three ANN stages were built for seventeen short-circuited fault localization. The proposed multistage architecture shows a performance increase of almost 25%.

Figure 5.10 resumes all of the work done during this thesis. Boxes with gray

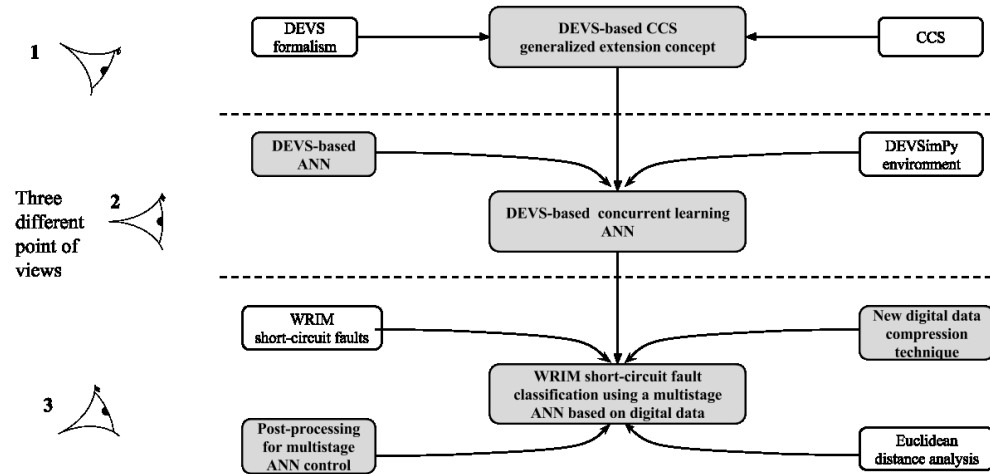


Figure 5.10 – Global view of the realized work.

shadow are the introduced concepts or the proposed architectures. It also shows two application stages, as well as three different points of view. The DEVS formalism and the CCS concept are used to create a new generic DEVS extension that offers a concurrent simulation without any changes in the classic DEVS simulator. The first application stage is to create a concurrent learning for ANNs. This is done using the DEVS-based CCS extension with the developed DEVS-based ANN inside the DEVSIMPy environment. The second level of application is to have a WRIM short-circuit fault classification using the concurrent learning multi-stage ANN. This is done using the Euclidean distance analysis with a new digital data compression technique and the developed multistage architecture.

After reading this thesis, the reader can achieve three different point of views (Figure 5.10). If the reader looks from the DEVS formalism perspective (view 1), then most probably the DEVS-based CCS is the most interesting part of the work, where a concurrent simulation can take place without changing the simulation kernel. To validate this concept, the concurrent ANN is presented. The second point of view comes from the ANN (view 2, Figure 5.10), where it is decomposed to several atomic models inside the DEVS formalism creating a new DEVS library. The presented modeling offers a more flexible ANN able to receive enhancements from the DEVS formalism. The concept was validated with the CCS simulation, giving runtime comparison between different configurations. The third point is presented from the WRIM perspectives, where it is important to detect the electrical faults as fast as possible with the lowest cost. At this point of view, the reader can be interested

into the electric currents digital compression, the neural network fault detection and localization for up to 16 faults, and the multistage architecture optimization.

After reading this thesis, the document order can be justified by the implementation orders where it begins with the DEVS-based CCS and ANN followed by the WRIM short-circuit diagnosis. The introduction presented the problem of the WRIM electrical fault detection and then unveiled the proposed solution track using the ANN, DEVS and the CCS. We preferred to well introduce the main application domain in the general introduction followed by the implementation in Chapters 2, 3 and 4 to finish with the results of the WRIM electrical fault detection in Chapter 5.

Perspectives

This thesis can be a new start to a multi-domain research. The new DEVS-based CCS is currently helpful for a single simulation path and with different values. Further studies have to take place in order to get the DEVS-based CCS to the next level of generalization where the DEVS messaging become more path independent. This idea can be realized by changing the classic DEVS messaging management (without changing the DEVS simulator) on the fly with respect for the object interface. In that manner the concurrent messaging can propagate disguised with the classic message interface. This idea bring a much wider range of applications and make easier for users and researchers a better tool to simulate concurrent behaviors.

The feed-forward artificial neural network architecture saw a new redesign where separation between feed-forward calculations and the learning algorithms makes it easier to alternate between different learning algorithms. As presented the DEVS-based ANN benefit from a concurrent DEVS extension. Further work might attack different DEVS extensions as Fuzzy-DEVS or PDEVS. Not only this model fragmentation is useful but also the new multistage building technique based on statistical analysis. Bringing the multistage architecture applied on different domains as pattern recognition and data prediction is a future. Automating the process between the statistical analysis and building the ANN architecture also is considered.

The electric motor fault detection is the main application of this thesis, which was the inspiration for all the work done, different ideas about future work can be

proposed. First, the multistage ANN validation on a unique fault detection, where neural networks learn only unique faults (only one short-circuit) and be able to detect combined faults (multiple short-circuits on both sides: rotor and stator). Second, testing this proposed pre-processing technique on the transient state of the machine and how well it can perform. Third, the implementation of the proposed ANN work on a FPGA board. The FPGA would allow to the ANN a hardware implementation, where the learning process is created on a PC with the concurrent DEVS-based ANN and then exported directly to the FPGA to be ready for an automatic short-circuit detection on the WRIMs. A first step towards this implementation already began with a study of the Nexys 3 FPGA board. As the FPGA boards are easily compatible with the VHDL programming language might create the challenge of the incomparability of the float numbers. This makes this thesis an open ground for a lot of future work in multiple fields.

Chapter 6

Annexes

Dendrogram DEVS plug-in

```
1 import wx
2 from matplotlib.pyplot import show from hcluster import pdist,
3 linkage, dendrogram
4
5 def OnLeftDClick(self, event)
6     """ Left Double Click has been invoked.
7     This plugin call pdist function from hcluster package and plot
8     the dendrogram using matplotlib.pyplot package. """
9     model = canvas.getCurrentShape(event)
10    devs = self.getDEVSMODEL()
11    if devs:
12        Y = pdist(devs.vectors)
13        Z = linkage(Y)
14        dendrogram(Z)
15        show()
16    else:
17        wx.MessageBox(_("No DEVS model is instanciaded.Go back to the simulation!"),
18            _("Info"), wx.OK|wx.ICON_INFORMATION)
```

Signature Class

```
1 class Signature(object):
2     def __init__(self, *args, **kwargs):
3         for key,value in kwargs.items():
4             setattr(self, key, value)
5         for val in args:
6             if isinstance(val, dict):
7                 for k,v in val.items():
8                     setattr(self, k, v)
9             elif isinstance(val, list):
10                for c in filter(lambda a: isinstance(a, (tuple,list)), val):
11                    setattr(self, c[0], c[1])
```

Input Layer Atomic Model

```

1 class Input(DomainBehavior):
2     """ Input Layer model
3         2 input ports:
4         - Learning input
5         - Validation input
6         Number of output depends of the input number
7     """
8     def __init__(self):
9         DomainBehavior.__init__(self)
10        self.state = {'status':'Idle', 'sigma': INFINITY}
11        self.current_tpattern = 0
12        self.current_vpattern = 0
13        self.t_pattern = []
14        self.v_pattern = []
15    def extTransition(self):
16        for port, msg in self.peek_all():
17            i = port.myID
18            if i == 0:
19                self.t_pattern.append(map(float,msg.value[0]))
20                self.dt = 1.0/len(self.t_pattern)
21                self.state = {'status':'PASSIVE', 'sigma':self.dt}
22                self.msgListOut =
23                    [Message([None,None],0.0) for i in xrange(len(self.OPorts))]
24            else:
25                self.v_pattern.append(map(float,msg.value[0]))
26    def outputFnc(self):
27        if self.state['status'] == 'ACTIVE':
28            for i in xrange(len(self.t_pattern[self.current_tpattern])):
29                tval = self.t_pattern[self.current_tpattern][i]
30                vval = self.v_pattern[self.current_vpattern][i] if self.v_pattern != []
31                    else None
32                msg = self.msgListOut[i]
33                msg.value = [tval,vval,self.myID]
34                self.poke(self.OPorts[i], msg)
35    def intTransition(self):
36        if self.state['status'] == 'PASSIVE':
37            self.state = {'status':'ACTIVE', 'sigma':0.0}
38        elif self.state['status'] == 'ACTIVE':
39            self.current_tpattern += 1
40            if self.current_tpattern >= len(self.t_pattern):
41                self.current_tpattern = 0
42            self.current_vpattern += 1
43            if self.current_vpattern >= len(self.v_pattern):
44                self.current_vpattern = 0
45            self.state = {'status':'PASSIVE', 'sigma':self.dt}
46    def concTransition(self):    if self.state['status'] == 'ACTIVE':

```

```
47     for sim in self.simData.simDico:
48         if not self.myID in self.simData.simDico[sim]:
49             s = self.createSim()
50             self.simData.addSim(sim,self.myID,s)
51             l = self.t_pattern[self.current_tpattern]
52             l_v = self.v_pattern[self.current_vpattern] if self.v_pattern != []
53                    else None
54             self.simData.simDico[sim][self.myID].outputs =
55                 dict(zip(xrange(len(l)),l))
56             self.simData.simDico[sim][self.myID].val_outputs =
57                 dict(zip(xrange(len(l_v)),l_v)) if self.v_pattern != [] else None
58 def createSim(self,dataInit={}):
59     dataInit['outputs'] = {}
60     dataInit['val_outputs'] = {}
61 def timeAdvance(self):
62     return self.state['sigma']
```

Hidden Layer Atomic Model

```

1 class Hidden(DomainBehavior):
2     """ Hidden Layer """
3     def __init__(self, bias = 1.0,
4                 N = 0.9,
5                 M = 0.1,
6                 activation_f = ("tanh","sigmoid"),
7                 k = 1.0,
8                 a = -0.2,
9                 b = 0.2,
10                fileName = os.path.join(os.getcwd(),"weights_%d"%randint(1,100)),
11                learning_flag = True):
12         """ constructor.
13             @param bias      = the bias value of hidden layer.
14             @param activation_function = the type of the activation function.
15             @param k        = param. for sigmoide activation function only.
16             @param a        = weight initialization range [a,b]
17             @param b        = weight initialization range [a,b]
18             @param fileName  = weights file
19             @param learning_flag = model status
20         """
21
22         DomainBehavior.__init__(self)
23         self.state = {'status':'Idle','sigma':INFINITY}
24         self.dataInit = {'N':N,'M':M,'bias':bias,'activation':activation_f[0]}
25         self.k = k
26         self.a = a
27         self.b = b
28         self.msgListOut = []
29         self.msgListIn = {}
30         self.sim = None
31         self.fileName = fileName
32         self.learning_flag = learning_flag
33         self.layerId = self.myID
34         seed(0)
35     def extTransition(self):
36         """ receiving only new input to claculate or
37             just a signal of weight changes. """
38         i=0
39         for port,msg in self.peek_all():
40             i = port.myID
41             self.msgListIn[i] = msg
42             value = msg.value
43             if i == (len(self.IPorts)-1):
44                 """ receiving weight changing signal """
45                 self.sim = value
46                 self.msgListIn = {}

```

```
47         break
48     else:
49         """ receiving new input to calculate """
50         if self.sim == None:
51             self.sim = Signature(self.createSim(self.dataInit))
52             self.sim.inputs[i] = value[0]
53             self.sim.val_inputs[i] = value[1]
54         if not i == (len(self.IPorts)-1):
55             self.transfer(self.sim)
56             self.state = {'status': 'ACTIVE', 'sigma': 0.0}
57     def concTransition(self):
58         if self.state['status'] == 'ACTIVE':
59             plid = self.msgListIn[0].value[2]
60             for Id in self.simData.simDico:
61                 try:
62                     sim = self.simData.simDico[Id]
63                     if not self.layerId in sim:
64                         self.simData.addSim(Id, self.layerId, self.createSim(self.dataInit))
65                         sim[self.layerId].previousId = plid
66                         sim[self.layerId].inputs.update(sim[plid].outputs)
67                 try:
68                     sim[self.layerId].val_inputs.update(sim[plid].val_outputs)
69                 except:
70                     pass
71                 self.transfer(sim[self.layerId])
72             except:
73                 pass
74     def outputFnc(self):
75         Nbports = len(self.OPorts)-1
76         for i in range(Nbports):
77             msg = self.msgListOut[i]
78             try:
79                 T = self.sim.outputs[i]
80                 V = self.sim.val_outputs[i]
81             except:
82                 V = None
83             msg.value= [T,V,self.layerId]
84             self.poke(self.OPorts[i],msg)
85         msg = self.msgListOut[Nbports]
86         msg.value = [self.sim, None, self.layerId]
87         self.poke(self.OPorts[Nbports],msg)
88         self.msgListIn = {}
89     def intTransition(self):
90         self.state = {'status': 'Idle', 'sigma': INFINITY}
91     def rand(self, a, b):
92         return (b-a)*random() + a
93     def timeAdvance(self):
94         return self.state['sigma']
```

```
95     def transfer(self,sim):
96         for i in sim.wh:
97             s = sum([sim.inputs[j]*sim.wh[i][j] for j in sim.wh[i]])
98             sim.outputs[i] = self.activation(sim.activation,s)
99             try:
100                 s_val = sum([sim.val_inputs[j]*sim.wh[i][j] for j in sim.wh[i]])
101                 sim.val_outputs[i] = self.activation(sim.activation,s_val)
102             except:
103                 sim.val_outputs[i]= None
104     def activation(self,activation,x,k=1.0):
105         """ activation Functions. """
106         if activation == "tanh":
107             return math.tanh(x)
108         elif activation == "sigmoid":
109             return 1.0/(1.0+math.exp(float(k)*(-float(x))))
110         else:
111             return x
112     def finish(self, msg):
113         """ optional method to control the
114         behavior when simulation finished """
115         pass
116     def createSim(self,dataInit):
117         dataInit['inputs'] = {len(self.IPorts)-1:self.dataInit['bias']}
118         dataInit['val_inputs'] = {len(self.IPorts)-1:self.dataInit['bias']}
119         dataInit['outputs'] = {}
120         dataInit['val_outputs'] = {}
121         dataInit['wh'] = {}
122         dataInit['c'] = {}
123         dataInit['errors'] = {}
124         dataInit['errorglobal'] = 0.0
125         dataInit['errorglobalvalidation'] = 0.0
126         dataInit['previousId'] = None
127         for i in range(len(self.OPorts)-1):
128             dataInit['wh'][i] = {}
129             dataInit['c'][i] = {}
130             for j in range(len(self.IPorts)):
131                 dataInit['wh'][i][j] = self.rand(self.a, self.b)
132                 dataInit['c'][i][j] = 0.0
133         self.msgListOut = [Message([None,None],0.0) for i in xrange(len(self.OPorts))]
134         return dataInit
135     def __str__(self):
136         return 'Hidden'
```

ErrorGenerator Atomic Model

```

1 class ErrorGenerator(QuickScope):
2     def __init__(self, stop_learning_factor = 0.0,
3                 stop_error_factor = 0.0, fusion = True, eventAxis = False):
4         """ Constructor
5             @param stop_learning_factor = descri1
6             @param stop_error_factor = descr2
7         """
8         QuickScope.__init__(self, fusion = True, eventAxis = False)
9         self.current_pattern = 0
10        self.current_validation_pattern = 0
11        self.iteration = 0
12        self.validation_iteration = 0
13        self.input_list = {}
14        self.input_list_validation = {}
15        self.errors = {}
16        self.errors_validation = {}
17        self.output_targets = []
18        self.output_validation_targets = []
19        self.globalerror = 0.0
20        self.globalerror_validation = 0.0
21        self.layerId = None
22    def extTransition(self):
23        """ recieving """
24        for port,msg in self.peek_all():
25            i = port.myID
26            if i>1:
27                value = msg.value
28                self.layerId = value[2]
29                self.input_list[i-2] = value[0]
30                self.input_list_validation[i-2] = value[1]
31            elif i == 0:
32                self.output_targets.append(map(float,msg.value[0]))
33            else:
34                self.output_validation_targets.append(map(float,msg.value[0]))
35        if i > 1:
36            results = self.errorCalc(self.input_list,self.input_list_validation)
37            self.globalerror += results['gErrors']
38            self.globalerror_validation += results['gErrors_v']
39            self.errors = results['errors']
40            self.state = {'status': 'ACTIVE' , 'sigma': 0.0}
41    def concTransition(self):
42        if self.state['status'] == 'ACTIVE':
43            for Id in self.simData.simDico:
44                ''
45                extracting the signature of the output layer
46                to get the output and calculate the errors

```

```
47     '''
48     try:
49         sim = self.simData.simDico[Id][self.layerId]
50         results = self.errorCalc(sim.outputs,sim.val_outputs)
51         sim.errors = results['errors']
52         sim.errorglobal += results['gErrors']
53         sim.errorglobalvalidation += results['gErrors_v']
54         if self.current_pattern == len(self.output_targets)-1:
55             try:
56                 self.results['t'+str(Id)].append((self.iteration,sim.errorglobal))
57             except:
58                 self.results['t'+str(Id)] = [(self.iteration,sim.errorglobal)]
59                 sim.errorglobal = 0.0
60         if self.current_validation_pattern == len(self.output_validation_targets)-1:
61             try:
62                 self.results['v'+str(Id)].append((self.iteration,sim.errorglobalvalidation))
63             except:
64                 self.results['v'+str(Id)] = [(self.iteration,sim.errorglobalvalidation)]
65                 sim.errorglobalvalidation = 0.0
66         except:
67             pass
68     def outputFnc(self):
69         self.poke(self.OPorts[0], Message([self.errors,None,None], self.timeNext))
70     def intTransition(self):
71         if self.current_pattern == len(self.output_targets)-1:
72             try:
73                 self.results['t'].append((self.iteration,self.globalerror))
74             except:
75                 self.results['t'] = [(self.iteration,self.globalerror)]
76             self.iteration += 1
77             self.current_pattern = 0
78             self.globalerror = 0.0
79         else:
80             self.current_pattern += 1
81         if self.current_validation_pattern == len(self.output_validation_targets)-1:
82             try:
83                 self.results['v'].append((self.iteration,self.globalerror_validation))
84             except:
85                 self.results['v'] = [(self.iteration,self.globalerror_validation)]
86             self.validation_iteration += 1
87             self.current_validation_pattern = 0
88             self.globalerror_validation = 0.0
89         else:
90             self.current_validation_pattern += 1
91         self.state = {'status':'Idle', 'sigma':INFINITY}
92     def timeAdvance(self):
93         return self.state['sigma']
94     def errorCalc(self,in_lst,in_lst_v):
```

```
95     errors = {}
96     errors_v = {}
97     a = 0.0
98     b = 0.0
99     cp = self.current_pattern
100    for i in in_lst:
101        errors[i] = float(self.output_targets[cp][i] - in_lst[i])
102        a = a + 0.5*(errors[i]*errors[i])
103        if self.output_validation_targets != []:
104            errors_v[i] = float(self.output_validation_targets[cp][i]-in_lst_v[i])
105            b = b+ 0.5*(errors_v[i]*errors_v[i])
106    return {'errors':errors, 'gErrors':a, 'gErrors_v':b}
```

DeltaWeight DEVS Atomic Model

```

1 class DeltaOutput_Weight(DomainBehavior):
2     def __init__(self):
3         """ constructor. """
4         DomainBehavior.__init__(self)
5         self.state = { 'status': 'Idle', 'sigma':INFINITY}
6         self.layerId = None
7         self.outError = {}
8         self.sim = None
9         self.msgListOut = [Message([None,None,None],0.0),Message([None,None,None],0.0)]
10    def extTransition(self):
11        """
12        receiving Errors and calculates new weights
13        """
14        for port,msg in self.peek_all():
15            i = port.myID
16            msg = self.peek(self.IPorts[i])
17            if i == 0:
18                self.sim = msg.value[0]
19                self.layerId = msg.value[2]
20            else:
21                self.sim.errors = msg.value[0]
22                self.outError = self.run(self.sim)
23                self.state = {'status': 'ACTIVE', 'sigma':0}
24    def run(self,sim):
25        m = {}
26        deltas = {}
27        outError = {}
28        for j in sim.wh:
29            m[j] = self.dactivation(sim.outputs[j],sim.activation)
30            deltas[j] = m[j]* float(sim.errors[j])
31            for k in sim.wh[j]:
32                change = deltas[j]*sim.inputs[k]
33                try:
34                    outError[k] = outError[k] + deltas[j]*sim.wh[j][k]
35                except:
36                    outError[k] = deltas[j] * sim.wh[j][k]
37                sim.wh[j][k] = sim.wh[j][k]+sim.N*change + sim.M*sim.c[j][k]
38                sim.c[j][k] = change
39        return outError
40    def outputFnc(self):
41        if (len(self.outError)) != 0:
42            msgError = self.msgListOut[0]
43            msgError.value = [self.outError,None,None]
44            msgSim = self.msgListOut[1]
45            msgSim.value = self.sim
46            self.poke(self.OPorts[1], msgError)

```

```
47     self.poke(self.OPorts[0],msgSim)
48 def intTransition(self):
49     self.state = {'status': 'Idle', 'sigma':INFINITY}
50
51 def concTransition(self):
52     if self.state['status'] == 'ACTIVE':
53         for Id in self.simData.simDico:
54             try:
55                 sim = self.simData.simDico[Id][self.layerId]
56                 errors = self.run(sim)
57                 self.simData.simDico[Id][sim.previousId].errors = errors
58             except:
59                 pass
60 def timeAdvance(self):
61     return self.state['sigma']
62 def dactivation(self,y,function="sigmoid"):
63     if function == "tanh":
64         return 1.0 - (y*y)
65     elif function == "sigmoid":
66         return y-y*y
67     else:
68         return 1.0
```

List of Publication

S. Toma, L. Capocchi, G.-A. Capolino, "Wound Rotor Induction Generator Inter-Turn Short-Circuits Diagnosis Using a New Digital Neural Network", *IEEE Transactions on Industrial Electronics*, vol.60, no.9, pp.4043-4052, Sept. 2013.

S. Toma, L. Capocchi, G.-A. Capolino, "An Efficient Architecture of Multi-Stage Neural Network for Wound-Rotor Induction Generator Short-Circuit Fault Classification", *In Proceedings of the XXth International Conference on Electrical Machines (ICEM'2012)*, September 2-5, 2012, Marseille (France), pp. 1565-1571.

L. Capocchi, S. Toma, G.-A. Capolino, F. Fnaiech, A. Yazidi, "Wound-Rotor Induction Generator Short-Circuit Fault Classification Using a New Neural Network Based on Digital Data", *In Proceedings of the 8th IEEE International Symposium on Diagnostics for Electrical Machines, Power Electronics and Drives*, September 5-8, 2011, Bologna (Italy), ISBN 978-1-4244-9302-9, IEEE Catalog Number CFP11SDE-USB, 6 pages.

S. Toma, L. Capocchi, D. Federici, "A New DEVS-Based Generic Artificial Neural Network Modeling Approach", *In Proceedings of The 23rd European Modeling and Simulation Symposium (Simulation in Industry)*, Rome, Italy, September 12-14, 2011, pp. 351-356.

Bibliography

- [1] G.-A. Capolino. A comprehensive analysis of the current status in low voltage induction motor diagnosis. In *Proceedings of the International conference on electrical machines*, pages 595–602, 2000.
- [2] A. Bellini, F. Filippetti, C. Tassoni, and G.-A. Capolino. Advances in diagnostic techniques for induction machines. *Industrial Electronics, IEEE Transactions on*, 55(12):4109–4126, 2008.
- [3] M. El Hachemi Benbouzid. A review of induction motors signature analysis as a medium for faults detection. *Industrial Electronics, IEEE Transactions on*, 47(5):984–993, 2000.
- [4] S. Nandi, H.A. Toliyat, and Xiaodong Li. Condition monitoring and fault diagnosis of electrical motors-a review. *Energy Conversion, IEEE Transactions on*, 20(4):719–729, 2005.
- [5] Frede Blaabjerg, Zhe Chen, Remus Teodorescu, and Florin Iov. Power electronics in wind turbine systems. In *Proceedings of the Power Electronics and Motion Control Conference, 2006. IPEMC 2006. CES/IEEE 5th International*, volume 1, pages 1–11. IEEE, 2006.
- [6] Pinjia Zhang, Yi Du, T.G. Habetler, and Bin Lu. A survey of condition monitoring and protection methods for medium-voltage induction motors. *Industry Applications, IEEE Transactions on*, 47(1):34–46, 2011.
- [7] Hua Su and Kil-To Chong. Induction machine condition monitoring using neural network modeling. *Industrial Electronics, IEEE Transactions on*, 54(1):241–249, 2007.

- [8] J.F. Martins, V.F. Pires, and A.J. Pires. Unsupervised neural-network-based algorithm for an on-line diagnosis of three-phase induction motor stator fault. *Industrial Electronics, IEEE Transactions on*, 54(1):259–264, 2007.
- [9] C. Demian, G. Cirrincione, and G.-A. Capolino. A neural approach for the fault diagnostics in induction machines. In *IECON 02 [IEEE 2002 28th Annual Conference of the, Industrial Electronics Society]*, volume 4, pages 3372–3376 vol.4, 2002.
- [10] Fiorenzo Filippetti, Giovanni Franceschini, and Carla Tassoni. Neural networks aided on-line diagnostics of induction motor rotor faults. *Industry Applications, IEEE Transactions on*, 31(4):892–899, 1995.
- [11] Bo Li, M-Y Chow, Yodyium Tipsuwan, and James C Hung. Neural-network-based motor rolling bearing fault diagnosis. *Industrial Electronics, IEEE Transactions on*, 47(5):1060–1069, 2000.
- [12] Hua Su and Kil To Chong. Induction machine condition monitoring using neural network modeling. *Industrial Electronics, IEEE Transactions on*, 54(1):241–249, 2007.
- [13] Gael Salles, Fiorenzo Filippetti, Carla Tassoni, G Crellet, and Giovanni Franceschini. Monitoring of induction motor load by neural network techniques. *Power Electronics, IEEE Transactions on*, 15(4):762–768, 2000.
- [14] Bernard P Zeigler, Herbert Praehofer, Tag Gon Kim, et al. *Theory of modeling and simulation*, volume 19. John Wiley New York, 1976.
- [15] Laurent Capocchi, Jean François Santucci, B. Poggi, and C. Nicolai. DEVSimpPy: A collaborative python software for modeling and simulation of devs systems. In Sumitra Reddy and Samir Tata, editors, *Proceedings of the WETICE*, pages 170–175. IEEE Computer Society, 2011. <http://code.google.com/p/devsimpy/>.
- [16] Laurent Capocchi, Fabrice Bernardi, Dominique Federici, and Paul-Antoine Bisgambiglia. BFS-DEVS: A general devs-based formalism for behavioral fault simulation. *Simulation Modelling Practice and Theory*, 14(7):945 – 970, 2006.

- [17] Mutasem Khalil Sari Alsmadi, Khairuddin Bin Omar, and Shahrul Azman Noah. Back propagation algorithm: the best algorithm among the multi-layer perceptron algorithm. *IJCSNS International Journal of Computer Science and Network Security*, 9(4):378–383, 2009.
- [18] R Sathya and Annamma Abraham. Comparison of supervised and unsupervised learning algorithms for pattern classification. (*IJARAI*) *International Journal of Advanced Research in Artificial Intelligence*, 2(2).
- [19] N Abdul Hamid, N Mohd Nawawi, and Rozaida Ghazali. The effect of adaptive gain and adaptive momentum in improving training time of gradient descent back propagation algorithm on classification problems. In *Proceeding of the International Conference on Advanced Science, Engineering and Information Technology*, pages 178–184, 2011.
- [20] Yan Xiong, Li Wang, and Dawei Li. Training feedforward neural networks by pruning algorithm based on grey incidence analysis. In *Proceedings of the Sixth Brazilian Symposium on Intelligent Information Technology Application IITA'08*, volume 3, pages 535–539, dec. 2008.
- [21] I.G. Maglogiannis. *Emerging Artificial Intelligence Applications in Computer Engineering: Real World Ai Systems With Applications in Ehealth, Hci, Information Retrieval and Pervasive Technologies*. Frontiers in artificial intelligence and applications, v. 160. Ios PressInc, 2007.
- [22] B.M. Wilamowski. How to not get frustrated with neural networks. In *Proceedings of the IEEE International Industrial Technology (ICIT)*, pages 5–11, march 2011.
- [23] H. Beigy and M.R. Meybodi. Adaptation of parameters of BP algorithm using learning automata. In *Proceedings of the Sixth Brazilian Symposium on Neural Networks*, pages 24–31, 2000.
- [24] T Jayalakshmi and A Santhakumaran. Statistical normalization and back propagation for classification. *International Journal of Computer Theory and Engineering*, 3(1):1793–8201, 2011.

- [25] D. Hunter, Hao Yu, M.S. Pukish, J. Kolbusz, and B.M. Wilamowski. Selection of proper neural network sizes and architectures - a comparative study. *Industrial Informatics, IEEE Transactions on*, 8(2):228–240, may 2012.
- [26] Ernst G. Ulrich, Vishwani D. Agrawal, and Jack H. Arabian. *Concurrent and comparative discrete event simulation*. Kluwer, 1994.
- [27] Katalin Popovici and Pieter J Mosterman. *Real-time simulation technologies: principles, methodologies, and applications*. CRC Press, 2013.
- [28] Robert W Righter. *Wind energy in America: A history*. University of Oklahoma Press, 1996.
- [29] Ahmed F Zobaa and Ramesh C Bansal. *Handbook of renewable energy technology*. World Scientific, 2011.
- [30] Bahatti. Three phase induction motor interview @ONLINE, 2011. http://electricalpowerinterview.blogspot.com/2011_09_01_archive.html.
- [31] AC and DC motors @ONLINE, August. 2013. <http://www.techtransfer.com/resources/wiki/entry/3726/>.
- [32] P.M.L. Santillan and M.M. Villena. Electromagnetic compatibility study in renewable energy induction generator. In *Power Tech Proceedings, 2001 IEEE Porto*, volume 1, pages 6 pp. vol.1–, 2001.
- [33] Peter Vas. *Parameter Estimation, Condition Monitoring, and Diagnosis of Electrical Machines (Monographs in Electrical and Electronic Engineering)*. London, UK: Oxford Univ. Press, 1993.
- [34] Greg Stone, Edward A Boulter, Ian Culbert, and Hussein Dhirani. *Electrical insulation for rotating machines: design, evaluation, aging, testing, and repair*, volume 21. Wiley. com, 2004.
- [35] P. F. Albrecht, J. C. Appiarius, R. M. McCoy, E.L. Owen, and D. K. Sharma. Assessment of the reliability of motors in utility applications - updated. *Energy Conversion, IEEE Transactions on*, EC-1(1):39–46, 1986.

- [36] Shahin Hedayati Kia, Humberto Henao, and G.-A. Capolino. Some digital signal processing techniques for induction machines diagnosis. In *Proceedings of the Diagnostics for Electric Machines, Power Electronics & Drives (SDEMPED), 2011 IEEE International Symposium on*, pages 322–329. IEEE, 2011.
- [37] Yasser Gritli, Luca Zarri, Claudio Rossi, Fiorenzo Filippetti, G.-A. Capolino, and Domenico Casadei. Advanced diagnosis of electrical faults in wound-rotor induction machines. *Industrial Electronics, IEEE Transactions on*, 60(9):4012–4024, 2013.
- [38] Alberto Bellini, Amine Yazidi, Fiorenzo Filippetti, Claudio Rossi, and G.-A. Capolino. High frequency resolution techniques for rotor fault detection of induction machines. *Industrial Electronics, IEEE Transactions on*, 55(12):4200–4209, 2008.
- [39] Alberto Bellini. Quad demodulation: A time-domain diagnostic method for induction machines. *Industry Applications, IEEE Transactions on*, 45(2):712–719, 2009.
- [40] Martin Blodt, Jeremi Regnier, and Jean Faucher. Distinguishing load torque oscillations and eccentricity faults in induction motors using stator current wigner distributions. *Industry Applications, IEEE Transactions on*, 45(6):1991–2000, 2009.
- [41] Wei Zhou, Thomas G Habetler, and Ronald G Harley. Bearing fault detection via stator current noise cancellation and statistical control. *Industrial Electronics, IEEE Transactions on*, 55(12):4260–4269, 2008.
- [42] Hugh Douglas, Pragasen Pillay, and Alireza K Ziarani. A new algorithm for transient motor current signature analysis using wavelets. *Industry Applications, IEEE Transactions on*, 40(5):1361–1368, 2004.
- [43] Shahin Hedayati Kia, Humberto Henao, and G.-A. Capolino. Torsional vibration effects on induction machine current and torque signatures in gearbox-based electromechanical system. *Industrial Electronics, IEEE Transactions on*, 56(11):4689–4699, 2009.

- [44] François Auger, Patrick Flandrin, Paulo Gonçalves, and Olivier Lemoine. Time-frequency toolbox. *CNRS France-Rice University*, 1996.
- [45] M. Morisue and H. Koinuma. Neural network for digital applications. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 2189–2192 vol.3, May 1989.
- [46] M.L. Badgero. Digitizing artificial neural networks. In *IEEE International Conference on Neural Networks*, volume 6, pages 3986–3989 vol.6, Jun./Jul. 1994.
- [47] Christer Karlsson, Jaime Arriagada, and Magnus Genrup. Detection and interactive isolation of faults in steam turbines to support maintenance decisions. *Simulation Modelling Practice and Theory*, 16(10):1689–1703, 2008.
- [48] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.
- [49] D.O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis, 2002.
- [50] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, USA, 1 edition, Nov 1995.
- [51] B Yegnanarayana. *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.
- [52] N. Kandil, K. Khorasani, R.V. Patel, and V.K. Sood. Optimum learning rate for backpropagation neural networks. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering, 1993.*, pages 465–468 vol.1, 1993.
- [53] SB Kotsiantis, D Kanellopoulos, and PE Pintelas. Data preprocessing for supervised learning. *International Journal of Computer Science*, 1(2):111–117, 2006.
- [54] Kamakshi Lakshminarayan, Steven A. Harp, and Tariq Samad. Imputation of missing data in industrial databases. *Applied Intelligence*, 11(3):259–275, 1999.
- [55] Bart LM Happel and Jacob MJ Murre. Design and evolution of modular neural network architectures. *Neural networks*, 7(6):985–1004, 1994.

- [56] Albrecht Schmidt and Zuhair Bandar. A modular neural network architecture with additional generalization abilities for large input vectors. In *Proceedings on Artificial Neural Nets and Genetic Algorithms*, pages 35–39. Springer, 1998.
- [57] Gabriel A Wainer and Pieter J Mosterman. *Discrete-event modeling and simulation: theory and applications*. CRC Press, 2010.
- [58] Bernard P. Zeigler. *Guide to Modeling and Simulation of Systems of Systems - User's Reference*. Springer Briefs in Computer Science. Springer, 2013.
- [59] BernardP. Zeigler and HessamS. Sarjoughian. *Languages for Constructing DEVS Models*. Simulation Foundations, Methods and Applications. Springer London, 2013.
- [60] Federico Bergero and Ernesto Kofman. A vectorial devs extension for large scale system modeling and parallel simulation. *Simulation*, 90(5):522–546, 2014.
- [61] Hans Vangheluwe. The discrete event system specification (devs) formalism. Technical report, Technical report, 2001. <http://moncs.cs.mcgill.ca>, 2001.
- [62] Xiaobo Li, Hans Vangheluwe, Yonglin Lei, Hongyan Song, and Weiping Wang. A testing framework for devs formalism implementations. In *Proceedings on the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 183–188, San Diego, CA, USA, 2011. Society for Computer Simulation International.
- [63] F. Perez, Brian E. Granger, and John D. Hunter. Python: An ecosystem for scientific computing. *Computing in Science and Engineering*, 13(2):13–21, 2011.
- [64] N. Rappin and R. Dunn. *WxPython in action*. Manning, 2006.
- [65] Eric Jones, Travis Oliphant, and Pearu Peterson. Scipy: Open source scientific tools for python, 2001. <http://www.scipy.org/>.
- [66] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9:10–20, 2007.

- [67] Mamadou K. Traoré and Alexandre Muzy. Capturing the dual relationship between simulation models and their context. *Simulation Modelling Practice and Theory*, 14(2):126 – 142, 2006.
- [68] V. Albert, A. Nketsa, and C. Seguin. Verifying trace inclusion between an experimental frame and a model. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 145:1–145:8, San Diego, CA, USA, 2010. Society for Computer Simulation International.
- [69] Byeong Soo Kim, Sun Ju Lee, Tag Gon Kim, and Hae Sang Song. Mapreduce based experimental frame for parallel and distributed simulation using hadoop platform. In *28th European Conference on Modelling and Simulation, ECMS 2014, Brescia, Italy, May 27-30, 2014*, pages 664–669, 2014.
- [70] Laurent Capocchi, Fabrice Bernardi, Dominique Federici, and Paul-Antoine Bisgambiglia. BFS-DEVS: A general devs-based formalism for behavioral fault simulation. *Simulation Modelling Practice and Theory*, 14(7):945 – 970, 2006.
- [71] High time for high-level test generation. In *Proceedings on the Panel at the IEEE International Test Conference*, pages 1112 – 1119, 1999.
- [72] Hessam S Sarjoughian, François E Cellier, and Bernard P Zeigler. *Discrete event modeling and simulation technologies: a tapestry of systems and AI-based theories and methodologies*. Springer, 2001.
- [73] Fernando J. Barros, Bernard P. Zeigler, and Paul A. Fishwick. Multimodels and dynamic structure models: an integration of dsde/devs and oopm. In *Proceedings of the 30th conference on Winter simulation, WSC '98*, pages 413–420, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.
- [74] L. Capocchi. *Simulation concurente de fautes comportementales pour des systèmes à évènements discrets: application aux circuits digitaux*. 2005. http://books.google.fr/books?id=p8c_XwAACAAJ.
- [75] Norbert Giambiasi, Bruno Escude, and Sumit Ghosh. Gdevs: A generalized discrete event specification for accurate modeling of dynamic systems. In *Proceed-*

- ings of the 5th International Symposium on Autonomous Decentralized Systems, 2001.*, pages 464–469. IEEE, 2001.
- [76] Tobias Schwatinski and Thorsten Pawletta. An advanced simulation approach for parallel devs with ports. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 147:1–147:8, San Diego, CA, USA, 2010. Society for Computer Simulation International.
- [77] John Vlissides, R Helm, R Johnson, and E Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49:120, 1995.
- [78] Wxformbuilder source code @ONLINE. December 2013. <http://sourceforge.net/projects/wxformbuilder/>.
- [79] Randall S Peterson, D Brent Smith, Paul V Martorana, and Pamela D Owens. The impact of chief executive officer personality on top management team dynamics: one mechanism by which leadership affects organizational performance. *Journal of Applied Psychology*, 88(5):795, 2003.
- [80] Michael Greenacre and Jorg Blasius. *Multiple correspondence analysis and related methods*. CRC Press, 2006.
- [81] Robert D. Short and K. Fukunaga. The optimal distance measure for nearest neighbor classification. *IEEE Transactions on Information Theory*, 27(5):622–627, Sep 1981.
- [82] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.
- [83] H. Marzi. Reconfigurable multi-stage neural networks in monitoring industrial machines. In *Proceedings of the 2005 IEEE Mid-Summer Workshop on Soft Computing in Industrial Applications, 2005. SMCia/05.*, pages 142–147, June 2005.
- [84] Shuang Yang, Antony Browne, and PhilipD. Picton. Multistage neural network ensembles. In Fabio Roli and Josef Kittler, editors, *Multiple Classifier Systems*, volume 2364 of *Lecture Notes in Computer Science*, pages 91–97. Springer Berlin Heidelberg, 2002.

- [85] H. Baltzakis and N. Papamarkos. A new signature verification technique based on a two-stage neural network classifier. *Engineering Applications of Artificial Intelligence*, 14(1):95 – 103, 2001.
- [86] Chi-hau Chen, Louis-François Pau, and Patrick Shen-pei Wang. *Handbook of pattern recognition and computer vision*. World Scientific, 2010.
- [87] William F Egan. *Frequency synthesis by phase lock*. Wiley New York, 2000.
- [88] Henrik O Johansson. A simple precharged cmos phase frequency detector. *Solid-State Circuits, IEEE Journal of*, 33(2):295–299, 1998.
- [89] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.
- [90] J. F. Mas and J. J. Flores. The application of artificial neural networks to the analysis of remotely sensed data. *International Journal of Remote Sensing*, 29(3):617, 2008.
- [91] S. Toma, L. Capocchi, and G.-A Capolino. Wound-rotor induction generator inter-turn short-circuits diagnosis using a new digital neural network. *IEEE Transactions on Industrial Electronics*, 60(9):4043–4052, Sept 2013.
- [92] R Development Core Team et al. R: A language and environment for statistical computing. *R foundation for Statistical Computing*, 2005.

Abstract

This thesis deals with the time-domain analysis of the electrical machines fault diagnosis due to early short-circuits detection in both stator and rotor windings. It also introduces to the Discrete Event system Specification (DEVS) a generic solution to enable concurrent and comparative simulations (CCS). The DEVS-based CCS is an extension introduced using an aspect-oriented programming (AOP) to interact with the classic DEVS simulator. A new DEVS-based ANN is also introduced with a separation between learning and calculation models. The DEVS-based CCS is validated on the proposed ANN DEVS library inside the DEVSimPy environment. The concurrent ANN contributes in the time-domains analysis for the electrical machine fault diagnosis. This new method is based on data coming directly from the sensors without any computation but with a new dedicated pre-processing technique. Later, some enhancements are brought to the artificial neural network based on a new multistage architecture reducing the training time and errors compared to the single ANN. The new architecture and techniques has been validated on real data sixteen non-destructive windings faults analysis and localization.