People’s Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
Abdelhafid Boussouf University Center Mila

Ref No.: . . . .

Institute of Mathematics and Computer Science
Department of Computer Science

Memory submitted in partial fulfillment of the requirements for the Master’s degree
Field: Computer Science

Specialty: Information and Communication Science and Technology (STIC)

Petri Nets and Deep Petri Net for Simulation, Modeling and Analysis of Complex Systems

Prepared by:
Djazi Qamar

Defended before the jury:

Bekhouche Maamar	MAB	Abdelhafid Boussouf Univ. Center, Mila	President
Hedjaz Sabrine	MAA	Abdelhafid Boussouf Univ. Center, Mila	Examiner
Aouag Mouna	MCB	Abdelhafid Boussouf Univ. Center, Mila	Rapporteur

Academic Year: 2024/2025

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Centre Universitaire Abdelhafid Boussouf Mila

Nř Ref : . . . .

Institut de Mathématiques et d'Informatique
Département d'Informatique

Mémoire préparé en vue de l'obtention du diplôme de Master
En : Informatique

Spécialité : Sciences et Technologies de l'Information et de la Communication (STIC)

Petri Nets and Deep Petri Net for Simulation, Modeling and Analysis of Complex Systems

Préparé par :
Djazi Qamar

Soutenue devant le jury :

Bekhouche Maamar	MAB	C. U. Abdelhafid Boussouf, Mila	Président
Hedjaz Sabrine	MAA	C. U. Abdelhafid Boussouf, Mila	Examinateur
Aouag Mouna	MCB	C. U. Abdelhafid Boussouf, Mila	Rapporteur

Année universitaire : 2024/2025

ACKNOWLEDGEMENTS

Above all, we thank **Allah** for granting us the strength, patience, and perseverance over the past five years, which enabled us to overcome challenges and bring this work to completion. Without His guidance and blessings, none of this would have been possible.

I would like to express my sincere gratitude to all those who have contributed to the successful completion of this work.

I am deeply grateful to my supervisor, **Mrs.Aouag Mouna**, for their invaluable guidance, continuous support, and constructive feedback throughout this research journey. Their insight, encouragement, and availability have been instrumental to the progress and completion of this work.

I also extend my warm thanks to the members of the jury for generously devoting their time to evaluating this memory and for offering valuable comments and suggestions that enriched its quality.

Special thanks are due to all the professors and administrative staff of the **Department of Computer Science** at the University of **Abdelhafid Boussouf**.

I am particularly thankful to my friends **Manar** and **Chahra** for their constant support, fruitful exchanges, and the motivating discussions that made this journey more enriching and enjoyable.

Finally, I would like to express my deepest appreciation to my family for their unconditional love, unwavering patience, and constant encouragement throughout these years. Their support has been a pillar of strength in both my academic and personal life.

Qamar

الإهداء

﴿ وَأَنْ لَيْسَ لِلْإِنْسَانِ إِلَّا مَا سَعَى(39) وَأَنَّ سَعْيَهُ سَوْفَ يُرَى ﴾

بعد توفيق من رب العالمين وبدعوة أم وأب، وبعد رحلة طويلة من الجد والاجتهاد وسهر الليالي، وبعد خيبات ومحاولات لا تنتهي ... ها قد حصدت تعب السنين الخمس

شكراً:

إلى من غرست في قلبي حب العلم والإصرار، إلى من كانت دعواتها رفيقة دربي، إلى من قدمت سعادتي وراحتي على سعادتها ... أمي

إلى الذي زين اسمي بأجمل الألقاب، من دعمني بلا حدود، إلى سندي وقوتي وملاذي ... أبي

إلى من وهبني الله نعمة وجودهم في حياتي، إلى العقد المتين، من كانوا عوناً في رحلتي، إخوتي ... "يعقوب، معتز بالله"

إلى خير ما أهدتني الأيام، توأَمَتْ روحي وفراشة قلبي ... راشا

إلى صديقات الخطوة الأولى والخطوة ما قبل الأخيرة، رفيقات الطفولة وأنيسات دربي ... "أماني، خلود"

إلى أجمل من قابلتني بهم الجامعة، رفيقات مشواري، داعماتي ومؤنساتي ... "إسراء، نهاد"

إلى عوض الله الجميل، من شاركتاني أصعب الليالي وأجمل الأيام في مشواري , رفيقات روحي ... "شهرة , منار"

إلى من كانت نصائحها و تشجيعها حافزا لإتمام هذا العمل مشرفتي " الأستاذة عواق منى "

الحمد لله الذي ما تم جهد ولا خُتِم سعي إلا بفضله

قمر

Abstract

The growing complexity of modern systems in fields such as artificial intelligence and large-scale simulations demands modeling approaches that are both expressive and formally analyzable. While Petri nets offer a solid foundation for modeling distributed systems, they fall short in representing adaptive behaviors typical of intelligent systems.

This work introduces two complementary approaches to bridge this gap. The first extends classical Petri nets into a new formalism called Deep Petri Nets, integrating learning mechanisms inspired by neural networks. Three metamodels are defined using Eclipse Modeling Framework (EMF): the first for Petri nets, the second for neural networks, and the third for Deep Petri Nets, combining structural features of both.

The second approach proposes a unified metamodel, NNPN (Neural Network of Petri Net), which merges Petri net and neural network elements into a cohesive model, later transformed into a Deep Petri Net.

Model transformations are achieved using Triple Graph Grammar (TGG) and Atlas Transformation Language (ATL) rules, ensuring consistency, traceability, and analytical capability. This dual-method framework supports the modeling of complex systems with both formal structure and adaptive behavior.

Keywords: Petri Nets, Neural Networks, Deep Petri Nets, Neural Network of Petri Net, Models Transformation, TGG, EMF and ATL.

Résumé

La complexité croissante des systèmes modernes dans des domaines tels que l'intelligence artificielle et les simulations à grande échelle exige des approches de modélisation à la fois expressives et formellement analysables. Bien que les réseaux de Petri offrent une base solide pour la modélisation des systèmes distribués, ils ne permettent pas de représenter les comportements adaptatifs typiques des systèmes intelligents.

Ce travail introduit deux approches complémentaires pour combler cette lacune. La première étend les réseaux de Petri classiques à un nouveau formalisme appelé réseaux de Petri profonds, intégrant des mécanismes d'apprentissage inspirés des réseaux neuronaux. Trois métamodèles sont définis à l'aide du Eclipse Modeling Framework (EMF): le premier pour les réseaux de Petri, le deuxième pour les réseaux neuronaux et le troisième pour les réseaux de Petri profonds, combinant les caractéristiques structurelles des deux.

La deuxième approche propose un métamodèle unifié, NNPN (Réseau de Neurones de Réseau de Petri), qui fusionne les éléments des réseaux de Petri et des réseaux neuronaux en un modèle cohésif, transformé ensuite en réseau de Petri profond.

Les transformations du modèle sont réalisées à l'aide des règles de la Triple Graph Grammar (TGG) et de l'Atlas Transformation Language (ATL), ce qui garantit la cohérence, la traçabilité et la capacité d'analyse. Ce cadre à double méthode soutient la modélisation de systèmes complexes avec une structure formelle et un comportement adaptatif.

Mots clés: Réseaux de Petri, Réseaux de Neurones, Réseaux de Petri Profonds, Réseau de Neuron de Réseaux de Petri, Transformation des Modèles, TGG, EMF et ATL.

الملخص

تتطلب التعقيدات المتزايدة للأنظمة الحديثة في مجالات مثل الذكاء الاصطناعي والمحاكاة على نطاق واسع أساليب نمذجة تكون تعبيرية وقابلة للتحليل الرسمي. بينما توفر شبكات بتري أساسًا قويًا لنمذجة الأنظمة الموزعة، فإنها تقصر في تمثيل السلوكيات التكيفية النموذجية للأنظمة الذكية.

يقدم هذا العمل نهجين تكميليين لسد هذه الفجوة. الأول يوسع الشبكات البترية الكلاسيكية إلى صيغة جديدة تُسمى الشبكات البترية العميقة، مدمجًا آليات التعلم مستوحاة من الشبكات العصبية. تم تعريف ثلاثة نماذج أولية باستخدام إطار عمل نمذجة Eclipse (EMF): الأول لشبكات بتري، الثاني للشبكات العصبية، والثالث لشبكات بتري العميقة، يجمع بين الميزات الهيكلية لكليهما.

النهج الثاني يقترح نموذج أولي موحد، NNPN (شبكة عصبية لشبكة بتري)، والذي يدمج عناصر شبكة بتري والشبكة العصبية في نموذج متماسك, يتم تحويله لاحقًا إلى شبكة بتري عميقة

يتم تحقيق تحويلات النموذج باستخدام قواعد الرسم البياني الثلاثي (TGG) ولغة تحويل أطلس (ATL), مما يضمن الاتساق والتتبع و القدرة التحليلية. يدعم هذا الإطار الثنائي طريقة نمذجة الأنظمة المعقدة ذات الهيكل الرسمي والسلوك التكيفي.

الكلمات المفتاحية: شبكات بتري، الشبكات العصبية , شبكات بتري العميقة، شبكة عصبية لشبكة بتري، تحويل النماذج، EMF TGG، و ATL.

Contents

List of Figures

List of Tables

General Introduction

Introduction

Petri nets are a powerful and widely used mathematical formalism for modeling, simulating, and analyzing systems in which concurrency, synchronization, and sequencing are fundamental. Introduced by Carl Adam Petri in 1962, they are valued for both their graphical clarity and rigorous theoretical foundation. Their applications span diverse domains, including computer science, telecommunications, industrial systems, and real-time control [1, 2]. Petri nets remain relevant in modern contexts such as cyber-physical systems, intelligent manufacturing, and business process management. [3–5]

In parallel, artificial neural networks (ANNs) have become a cornerstone of artificial intelligence, particularly through advances in deep learning. They excel in pattern recognition, prediction, and classification tasks. However, their internal mechanisms are often opaque, making them difficult to interpret and analyze formally. [6, 7]

In this work, we propose two hybrid modeling approaches that combine classical Petri nets, neural networks, and a novel formalism known as Deep Petri Nets (DPNs). This framework leverages the interpretability and formal rigor of Petri nets alongside the adaptive learning capabilities of neural networks. Deep Petri Nets aim to provide a unified model capable of both learning and adaptation, while preserving formal structure and analytical strength. [64] Model transformations are automated using the Triple Graph Grammar (TGG) approach where appropriate, and refined with ATL rules to ensure consistency, traceability, and flexibility of the hybrid models throughout the development lifecycle.

Problematic

Despite the widespread use of Petri nets for formally modeling and analyzing concurrent systems, and the success of neural networks in learning complex patterns and behaviors, a fundamental disconnect still exists between symbolic modeling techniques and data-driven learning approaches. Petri nets offer structure, traceability, and formal verification, but lack the ability to learn from data. Neural networks, while highly adaptive, are opaque and difficult to interpret or analyze formally. Bridging this gap is essential for designing intelligent systems that are both adaptive and explainable. This raises the following challenge: how can we develop a unified modeling approach that combines the analytical strengths of Petri nets with the learning capabilities of neural networks, while ensuring traceability, adaptability, and semantic consistency?

Contributions

To realize the proposed framework, we contribute two model-driven transformation approaches:
Approach 1: We define three distinct metamodels using the Eclipse Modeling Framework (EMF): the first for classical Petri nets, the second for neural networks, and the third for Deep Petri Nets (DPNs). The DPN metamodel integrates both the structural semantics of Petri nets and the learning capabilities of neural networks. Transformation rules are implemented to enable the systematic mapping of model elements between these metamodels, thereby facilitating the automated construction of Deep Petri Nets from heterogeneous models.

Approach 2: We design a unified intermediate metamodel called the Neural Network of Petri Net (NNPN), which embeds Petri net semantics directly into neural components. This hybrid representation serves as a bridge model that is subsequently transformed into a Deep Petri Net. This approach enables a tighter integration of learning dynamics within a structure that remains formally analyzable.

Memory Organization

We have organized this memory as follows: We begin with a general introduction that summarizes the content of our work.

- **Chapter 1:** This chapter introduces the foundational paradigms of our work: *Petri Nets (PNs)* for formal system modeling and *Neural Networks (NNs)* for adaptive learning. Their integration in *Deep Petri Nets (DPNs)* combines structural clarity with learning capabilities, enabling intelligent systems that are both explainable and dynamic.We also present key concepts of model transformation within the *Model-Driven Architecture (MDA)* framework, focusing on graph-based techniques such as *Triple Graph Grammars (TGGs)*. These enable automated and bidirectional synchronization between models, forming the backbone of our hybrid modeling approach.

- **Chapter 2:** This chapter provides a comprehensive review of related works, including foundational research and recent developments in the use of Petri Nets, neural networks, and hybrid models. We analyze various integration strategies such as fuzzy Petri Nets, neural Petri Nets, and deep Petri Nets. A comparative analysis highlights the strengths and limitations of existing approaches and positions our contribution within the current state of the art.

- **Chapter 3:** This chapter presents two complementary transformation-based approaches for generating Deep Petri Nets (DPNs) from classical Petri Nets. Both methods are implemented using the Atlas Transformation Language (ATL) and rely on metamodels defined in the Eclipse Modeling Framework (EMF). The first method involves a two-step transformation via an intermediate Neural Network model, while the second integrates Petri Net and Neural Network elements into a unified intermediate metamodel (NNPN). Detailed transformation rules, metamodels, and implementation processes are discussed.

- **Chapter 4:** In this chapter, we apply the proposed transformation approaches to two real-world intelligent systems: a Smart Traffic Management System and an Automated Production Workshop. These case studies demonstrate the practical benefits of our methodology for modeling and simulating complex and dynamic systems.

We end with a general conclusion that highlights its main achievements and outlines possible future works.

State of Art

Chapter1

Basic Concepts

**(Petri Net, Neural Network,
Deep Petri Net,
Model Transformaion)**

Introduction

As intelligent systems grow in complexity, combining symbolic modeling with data-driven learning becomes essential. This chapter introduces *Petri Nets (PNs)* for formal system modeling, *Neural Networks (NNs)* for adaptive learning, and their integration in *Deep Petri Nets (DPNs)* to support both structure and learning.

To enable this integration, we present core principles of model transformation within the Model-Driven Architecture (MDA) framework. We highlight graph-based techniques, especially *Triple Graph Grammars (TGGs)*, which offer formal, bidirectional synchronization between source and target models. This transformation layer is key to generating hybrid models in an automated and consistent way.

1.1 Petri Nets (PNs)

1.1.1 Definition

A Petri Net is a graphical and mathematical modeling tool used to describe and analyze the flow of information, resources, or control in concurrent, distributed, asynchronous, or stochastic systems. Originally introduced by Carl Adam Petri in 1962, it provides a formal framework for modeling discrete-event dynamic systems. [9].

Formally, a PN is defined as a 4-tuple (P, T, F, W), where:

- P is a finite set of **places**.
- T is a finite set of **transitions**.
- $F \subseteq (P \times T) \cup (T \times P)$ is the **flow relation**.
- $W : F \to \mathbb{N}^+$ is a **weight function**.

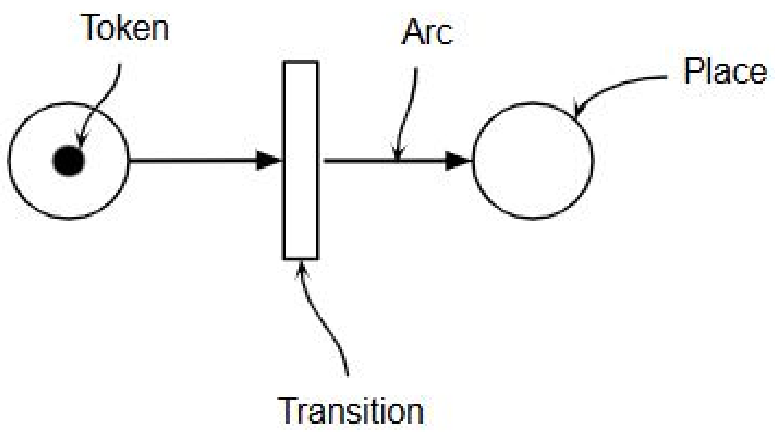


Figure 1.1: Components of Petri Net

Places and transitions are linked via directed arcs, subject to the following constraints:

- Each arc must connect a **place** to a **transition**, or a **transition** to a **place**. Connections between two places or between two transitions are not allowed.
- Every arc carries a positive integer **weight**. A weight of k indicates k parallel arcs, while an unlabeled arc is implicitly assigned a weight of 1.

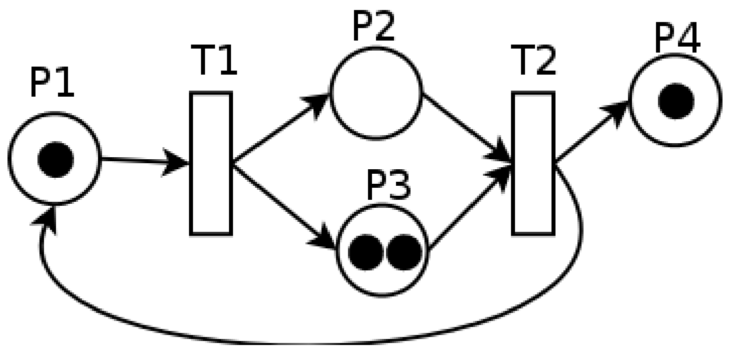


Figure 1.2: Petri Net

Basic Concepts of Petri Nets

In the context of Petri Nets, three core elements define the structure and behavior of a system [10]:

- **Place:** Represents a condition or a state in the modeled system. A place may contain tokens, and the distribution of tokens over places (called *marking*) describes the current state of the system.
- **Transition:** Represents an event or action that may change the state of the system. A transition can fire when certain conditions (such as token availability in input places) are met.
- **Pre-conditions and Post-conditions (Input and Output Arcs):** The arcs connecting places and transitions define the pre-conditions (input arcs) and post-conditions (output arcs) of transitions. When a transition fires, it consumes tokens from its input places and produces tokens in its output places, reflecting a change in the system state.

Marking in Petri Nets

In Petri nets, the **marking** represents the dynamic state of the system by describing how tokens are distributed across the places. Formally, a marking is defined as a function $M : P \to \mathbb{N}$, where each place $p \in P$ is mapped to a non-negative integer $M(p)$, indicating the number of tokens in that place [10]. The evolution of the marking over time, caused by the firing of transitions, models the behavior of the system.

The marking plays a central role in analysis techniques such as *reachability*, *boundedness*, *liveness*, and *invariants*, as it determines which transitions are enabled and how the state can evolve over time [11].

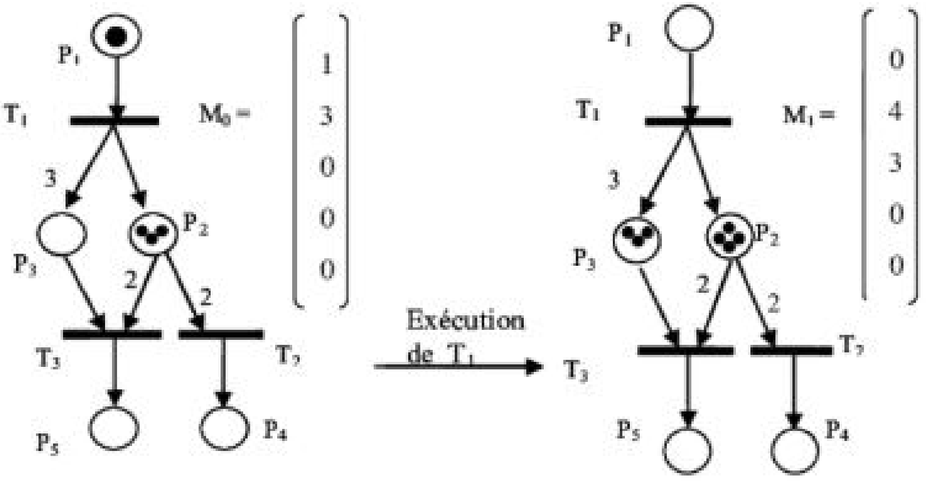


Figure 1.3: Marked Petri Net

1.1.2 Categories of Petri Nets

Petri nets can be categorized based on their structural properties and behavioral characteristics. Below are some common categories:

- **State graph :** A Petri net unmarked graph is an state graph if in which each transition has exactly one input place and one output place. [12]

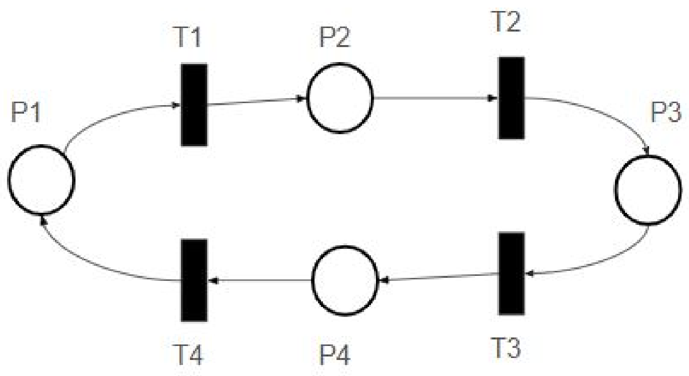


Figure 1.4: State Graph

- **Event Graph:** A Petri net where each place has exactly one incoming arc and one outgoing arc. [12]

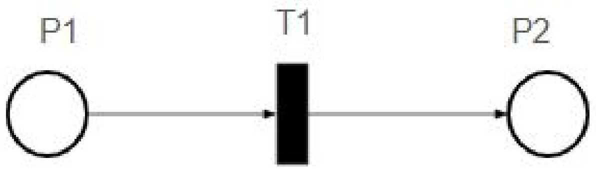
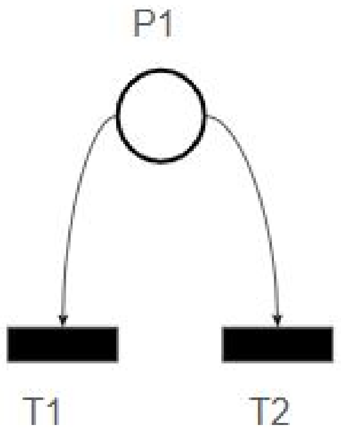
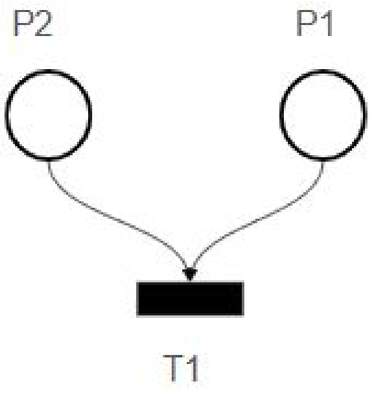


Figure 1.5: Event Graph

- **Conflict:**
  - **With Conflict:** Two or more transitions share the same input place, meaning that the firing of one transition disables the others.
  - **Without Conflict:** each place is connected to at most one outgoing transition. [1]



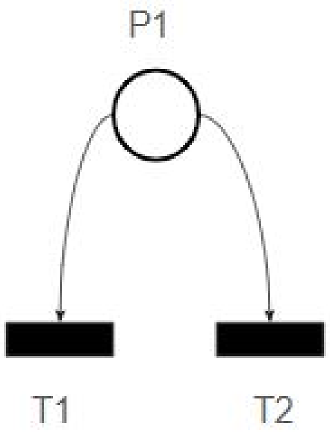
(a) With Conflict



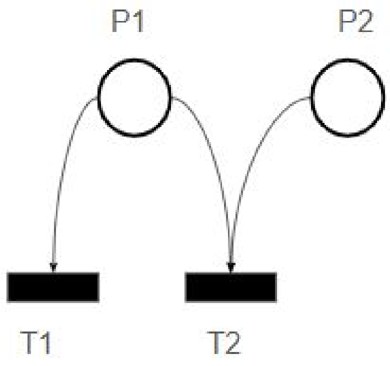
(b) Without Conflict

Figure 1.6: Comparison of Petri Nets: With and Without Conflict

- **Free Choice Net:** A Petri net where, whenever two or more transitions share an input place, that place is their only input. [13]



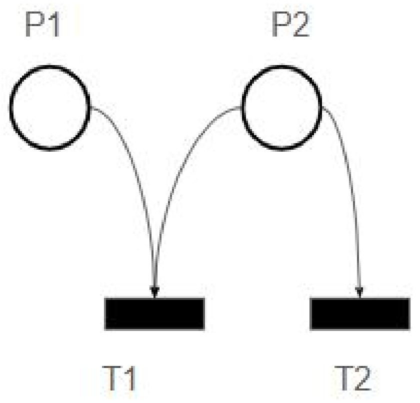
(a) Free Choice



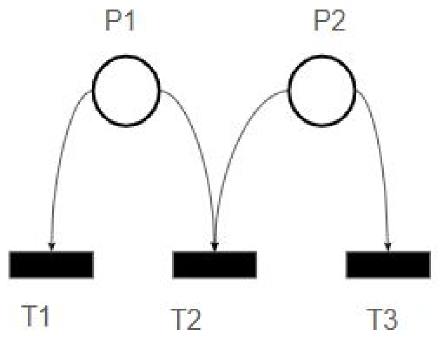
(b) Without Free Choice

Figure 1.7: Comparison of Petri Nets: With and Without Free Choice

- **Simple Petri Net:** A simple Petri net is a net in which each transition can be involved in at most one conflict. [9]



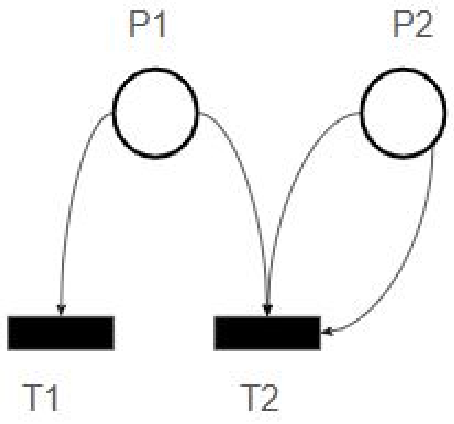
(a) Simple



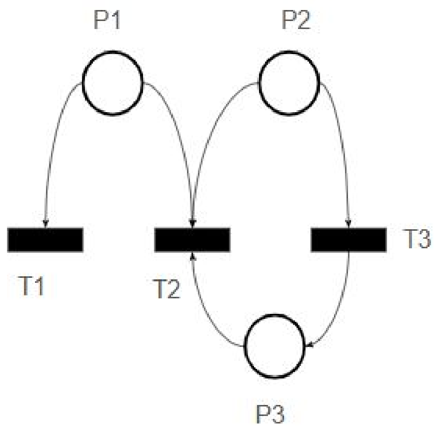
(b) Not Simple

Figure 1.8: Comparison of Petri Nets: Simple and Not Simple

- **Pure Petri Net:** A pure Petri net is a net in which there is no transition that has an input place which is also its output place. [1]



(a) Not Pure



(b) Pure

Figure 1.9: Comparison of Petri Nets: Pure and Not Pure

- **Generalized Petri Net:** A Petri net where strictly positive integer weights are associated with arcs. If an arc (P_i, T_j) has weight K, the transition T_j can fire only if place P_i contains at least K tokens; firing T_j then removes K tokens from P_i. Similarly, if an arc (T_j, P_i) has weight K, firing T_j adds K tokens to P_i. When no weight is indicated, it is assumed to be equal to 1 by default. [14]

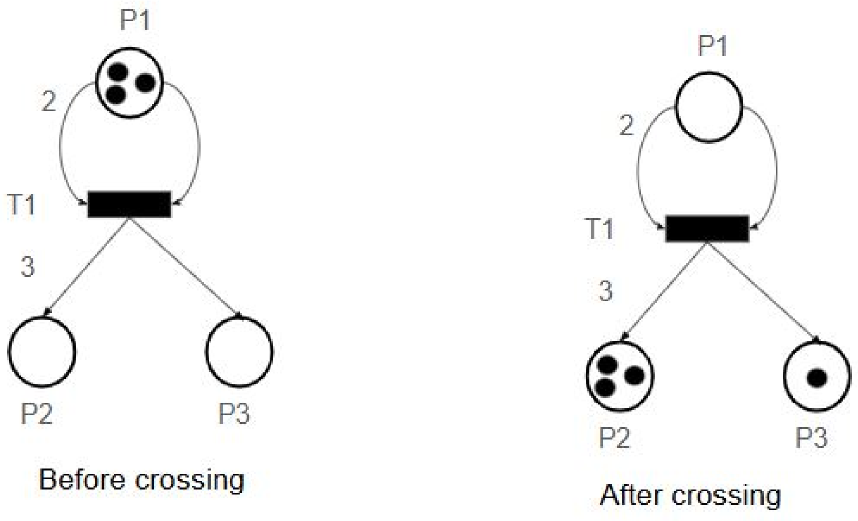


Figure 1.10: Generalized Petri Net

- **Capacity Petri Net:** A net where each place has an associated maximum number of tokens it can hold, known as its capacity. [1]

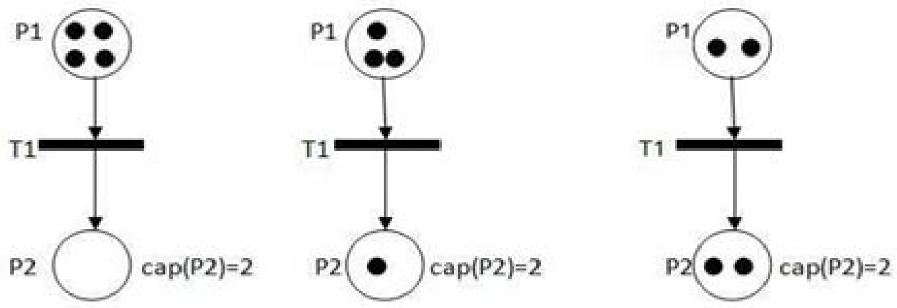


Figure 1.11: Capacity Petri Net

- **Priority Petri Net:** A net where transitions are assigned priorities. When multiple transitions are enabled simultaneously, only those with the highest priority are allowed to fire. [15]

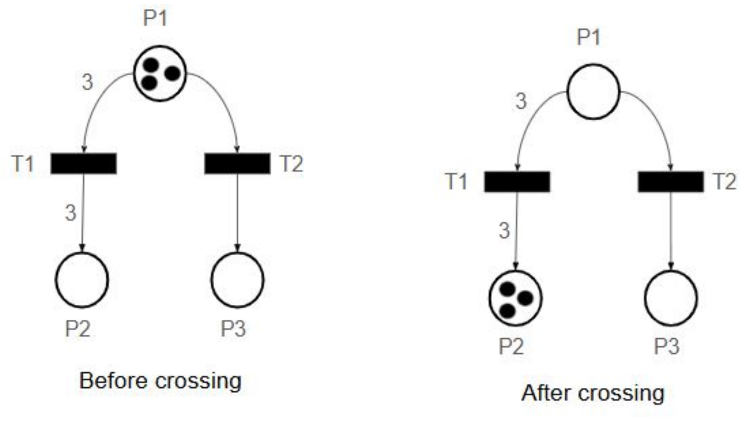


Figure 1.12: Priority Petri Net

- **loop-free Petri Net:** A net with no cycles in its underlying graph, meaning it is impossible to return to the same node (place or transition) by following a directed sequence of arcs. [9]

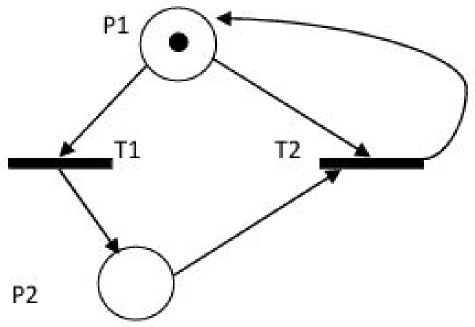


Figure 1.13: loop-free Petri Net

1.1.3 Petri Nets Properties

There are three fundamental properties of Petri nets.

- **Bounded :**
  - A place P_i is considered **bounded** for an initial marking M_0 if, for every marking reachable from M_0, the number of tokens in P_i remains finite.
  - A Petri Net is **bounded** for an initial marking M_0 if all places are bounded for M_0. Specifically, if for every marking M in the set of markings reachable from M_0 (denoted as M_0^*), we have $M(P_i) \leq K$, where K is a natural number, then P_i is said to be **K-bounded**. If this condition holds for every place, the Petri Net is termed **K-bounded**. [12]

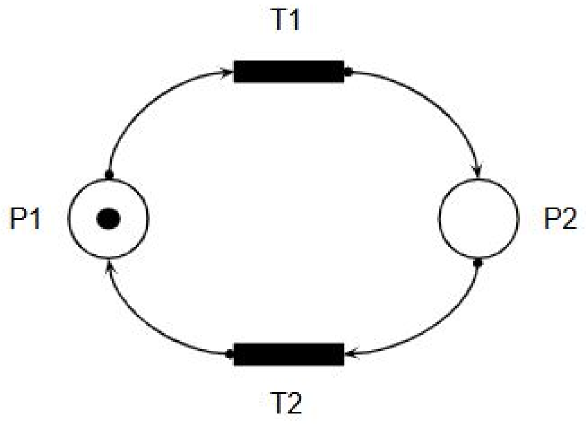
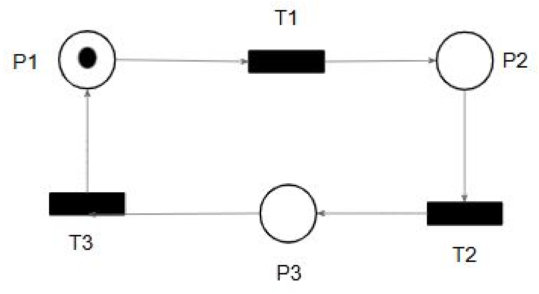


Figure 1.14: Bounded Petri Net

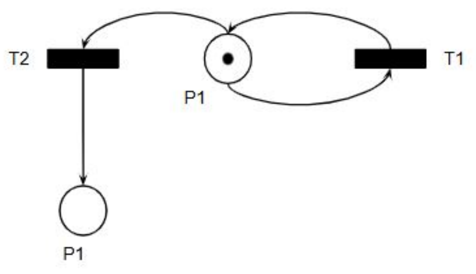
- **Liveness:**

  A Petri net is said to be live if, regardless of the marking reached during its execution, it is always possible to eventually fire any transition, possibly after a finite number of other transitions have occurred. [16]

  - **Quasi-Liveness :** Each transition can occur at least once in some firing sequence starting from the initial marking.



(a)



(b)

Figure 1.15: (a)Liveness Petri Net (b) Quasi-Liveness Petri Net

- **Blocking :**

  There exists at least one reachable marking from which no transition is enabled, meaning the system reaches a deadlock. [17]

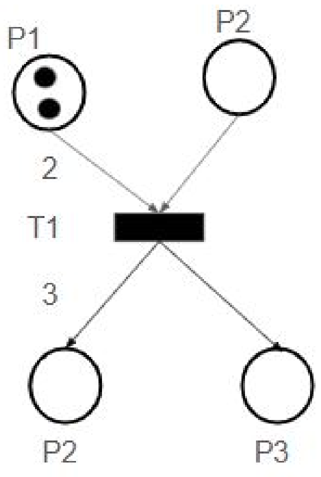


Figure 1.16: Blocking Petri Net

1.1.4 The Types of Petri Nets

Various types of Petri nets have been developed to model different kinds of systems and behaviors.

- **Timed Petri Net (TPN) :** is a type of Petri net where each transition is associated with a time delay, specifying the amount of time that must pass before the transition can fire. The timing aspect can be either deterministic or stochastic, and it allows the model to represent real-time behavior, such as the waiting time for events, synchronization, and delays in the system. [18].

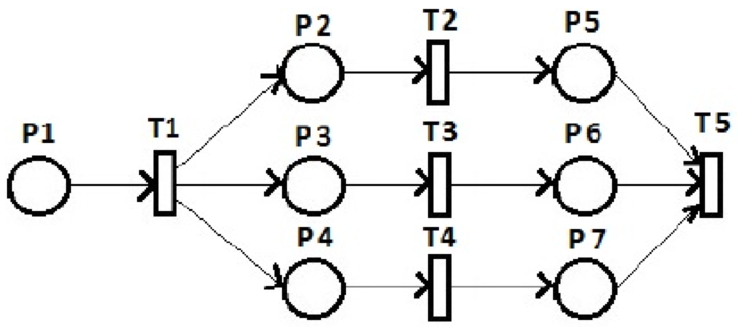


Figure 1.17: Timed Petri Net

- **Stochastic Petri Net (SPN) :**

  is a Petri net in which the firing times of transitions are governed by probabilistic distributions, typically exponentially distributed. This allows the net to model systems with random delays between events, such as those found in communication networks, manufacturing systems, or biological processes. [19].

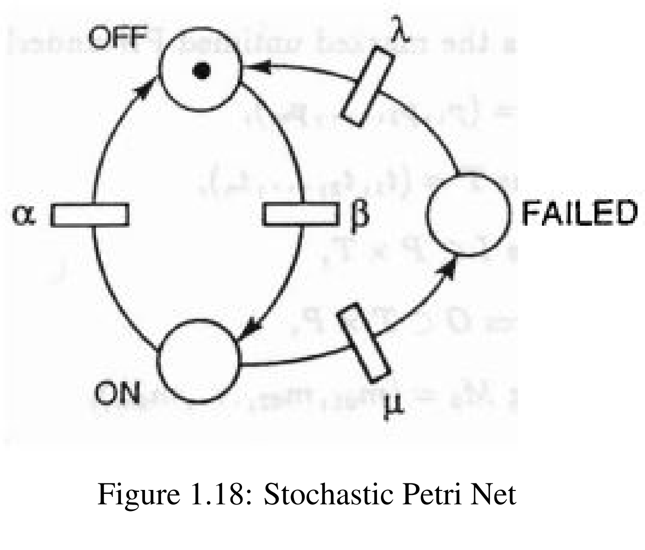


Figure 1.18: Stochastic Petri Net

- **Colored Petri Net (CPN) :**

  is a type of Petri net in which tokens have data values (known as colors), and transitions can operate on tokens according to the data they carry. In CPN, places can contain tokens of different types, and transitions can modify these tokens by changing their data values (colors). [20].

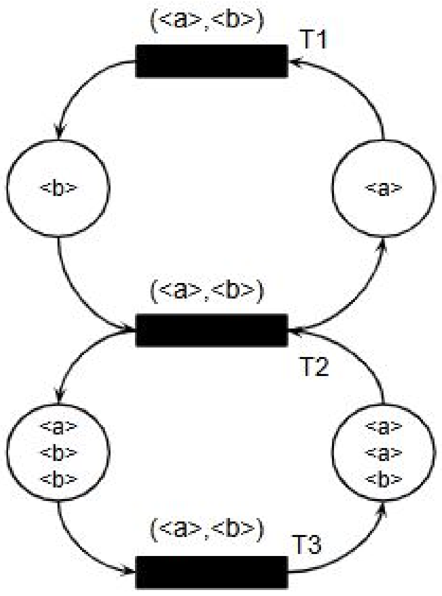


Figure 1.19: Colored Petri Net

1.2 Neural Networks (NNs)

1.2.1 Definition

A **Neural Network** is a computational system inspired by the brain's neural architecture. It consists of multiple layers of connected artificial neurons, each transforming input data through weighted combinations and non-linear activation functions. By iteratively adjusting these weights during training, the network learns to capture complex patterns in data, enabling it to perform tasks such as classification, regression, and pattern recognition. [21]

1.2.2 Mathematical Representation

Given inputs $x_1, x_2, \ldots, x_n$, corresponding weights $w_1, w_2, \ldots, w_n$, and a bias b, the neuron computes a weighted sum:

$$y = f(x) = w_0 + \sum_{d=1}^{D} w_d x_d \tag{1.1}$$

w_0 represents the intercept term, which corresponds to the weight of a hypothetical input $x_0 = +1$. [22]

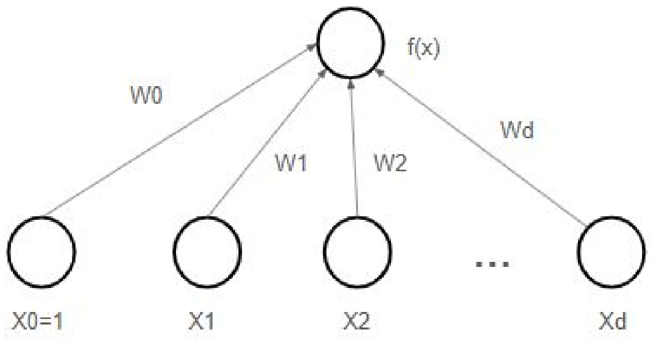


Figure 1.20: Neural Network

The Output Error:

The output error produced by a training example $(x_i, y_i), i \in \{1, \ldots, N\}$ is defined as follows: [22]

$$E_i(\mathbf{w}) = \frac{1}{2}(y_i - f(x_i))^2 \tag{1.2}$$

1.2.3 Neural Networks Architecture

A *Neural Network Architecture* refers to the structure or design of a neural network, describing the number of layers, the types of layers, and how the neurons (or nodes) in these layers are connected. The architecture determines the overall functionality and capacity of the network to learn and generalize from the data. [7]

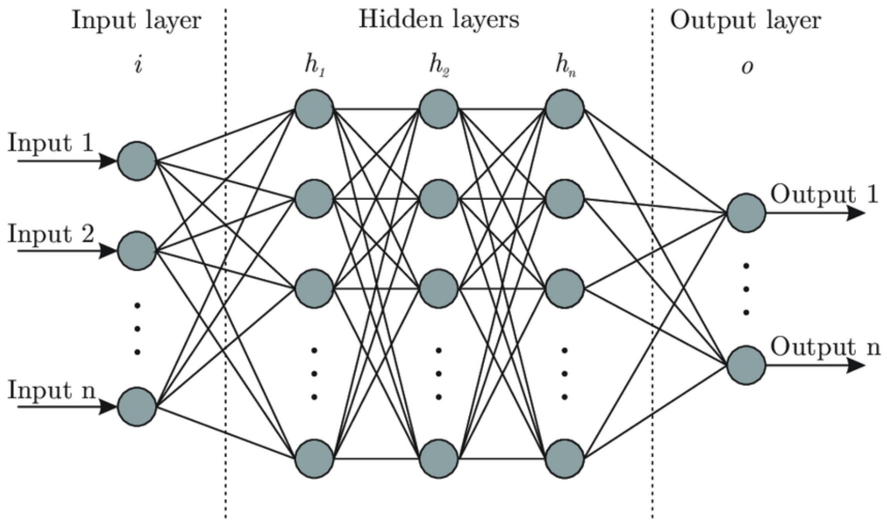


Figure 1.21: Neural Network Architecture

- **Input Layer :** the layer that receives the input features.
- **Hidden Layer :** These are *intermediate layers* where data is processed using various activation functions.
- **Output Layer :** The final layer that produces the network's predictions or classifications.
- **Connections (Weights) :** Each connection between neurons has a weight that is learned during training.
- **Activation Functions :** Nonlinear functions applied to the weighted sum of inputs at each neuron, introducing nonlinearity to the model Common activation functions include:
  - **ReLU (Rectified Linear Unit):** The ReLU function is defined as :

$$f(x) = \max(0, x) \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

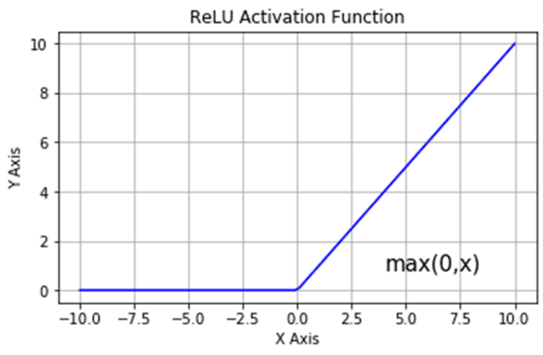


Figure 1.22: ReLU Activation Function

  - **Sigmoid:** The Sigmoid function is defined as :

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

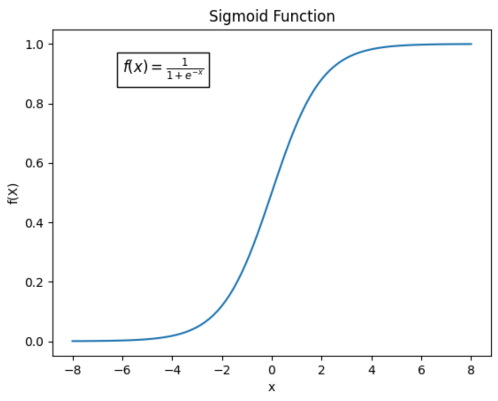


Figure 1.23: Sigmoid Activation Function

– **Tanh (Hyperbolic Tangent):**

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

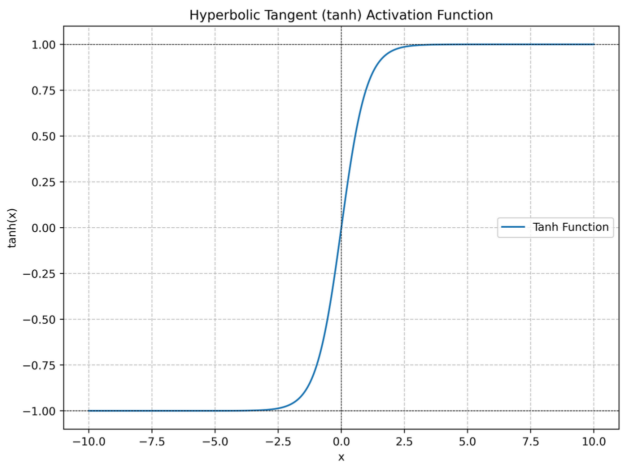


Figure 1.24: Tanh Activation Function

1.2.4 The Types of Neural Networks

Neural networks come in various architectures, each tailored to specific types of data and learning tasks. Below are some of the most common types:

- **Feedforward Neural Networks (FNNs)** The simplest form of neural networks where data flows in one direction from input to output without any cycles or loops. [23]

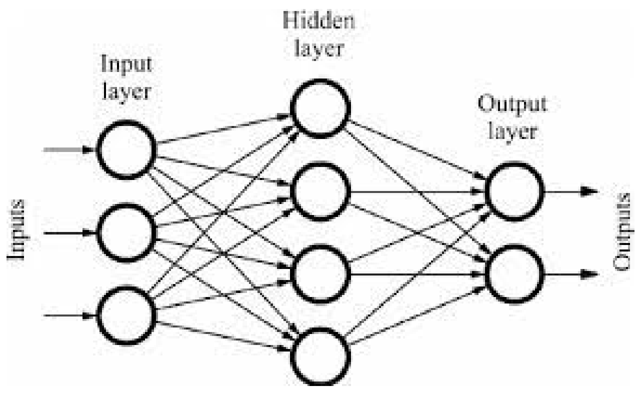


Figure 1.25: Feedforward Neural Networks

- **Convolutional Neural Networks (CNNs)** CNNs are specialized neural networks designed to process data with a grid like topology, such as images. They use convolutional layers to detect patterns and spatial hierarchies. [24]

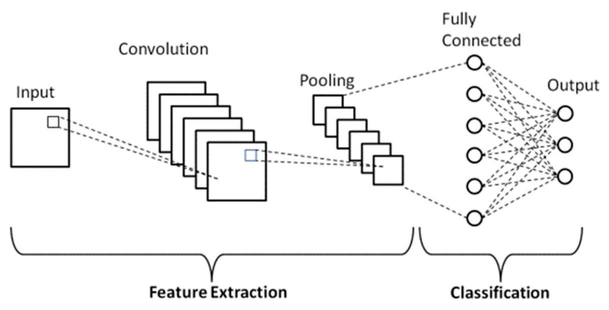


Figure 1.26: Convolutional Neural Networks

- **Recurrent Neural Networks (RNNs)** RNNs are neural networks that include loops, allowing information to persist across steps of input sequences. They are particularly suited for time-series data and natural language processing. [25]

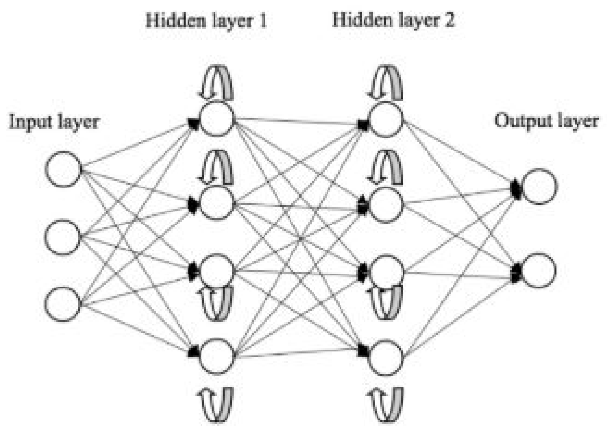


Figure 1.27: Recurrent Neural Networks

- **Generative Adversarial Networks (GANs)** GANs consist of two competing networks: a generator that creates data and a discriminator that evaluates it. Together, they produce increasingly realistic synthetic data. [26]

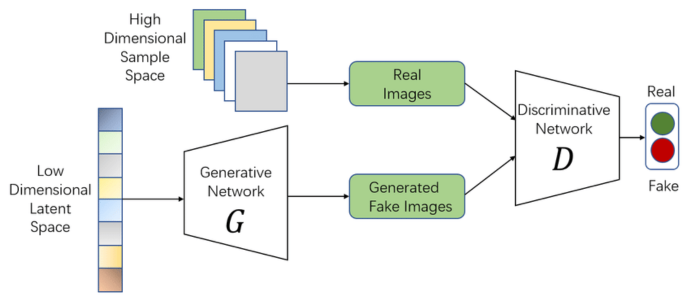


Figure 1.28: Generative Adversarial Networks

- **Graph Neural Networks (GNNs)** GNNs are designed to work with graph-structured data, capturing relationships between nodes and their neighbors. They are used in chemistry, social networks, and recommendation systems. [27]

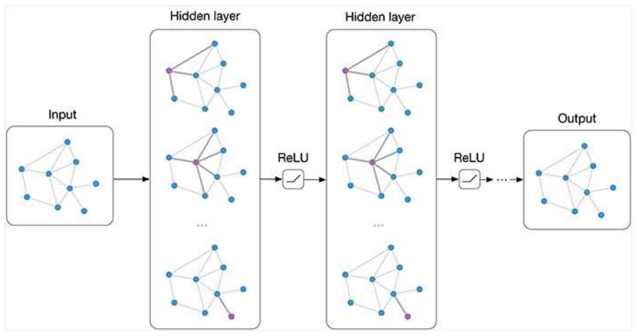


Figure 1.29: Graph Neural Networks

Remark. In our work, we employ the use of Feedforward Neural Networks (FNNs) as the underlying neural architecture due to their simplicity and effectiveness in modeling static input-output relationships.

1.3 Deep Petri Nets (DPNs)

1.3.1 Definition

Deep Petri Nets (DPNs) are an advanced extension of classical Petri nets that integrate learning mechanisms inspired by neural networks, while maintaining the formal structure and analytical power of traditional Petri nets. Introduced and formalized in recent works such as Lin et al. [28], DPNs aim to bridge the gap between symbolic modeling and data-driven machine learning by embedding layers of processing (akin to deep learning layers) into the Petri net structure.

According to Lin et al. [28], a Deep Petri Net is built on the foundation of High-Level Fuzzy Petri Nets (HLFPNs) and is designed to perform both supervised and unsupervised learning.

One of the defining features of DPNs is the presence of a supervisory node at the outermost layer. This node monitors parameter changes, helping track and control the learning process. This structural transparency enhances explainability, offering a clear advantage over traditional deep neural networks [29].

1.3.2 Formal Definition of A Deep Petri Nets

A Deep Petri Net (DPN) is defined as a 7-tuple:

$$\text{DPN} = (P, T, F, W, M_0, N, \Phi)$$

Where:

- P: A finite set of places.
- T: A finite set of transitions, such that $P \cap T = \emptyset$.
- F: A set of directed arcs (flow relation), where $F \subseteq (P \times T) \cup (T \times P)$.
- W: A weight function $W : F \to \mathbb{N}^+$ assigning a positive integer weight to each arc.
- M_0: The initial marking $M_0 : P \to \mathbb{R}_{\geq 0}$, which may be real-valued (enabling fuzzy or continuous token semantics).
- N: A set of neural networks $N = \{N_t \mid t \in T\}$, each N_t associated with a transition t.
- Φ: A function $\Phi : T \to \mathbb{R}$ that assigns a firing threshold or condition to each transition . [31]

1.3.3 Transition Firing Mechanism and Dynamic Behavior

The firing of a transition in a Deep Petri Net (DPN) involves two key computations: the activation output and the marking update. [30]

- **Activation Output** $\phi(t)$: The activation output controls how strongly a transition fires.

$$\phi(t) = A_t\left(\sum_{p \in \bullet t} W(p,t) \cdot M(p) + \sigma(t)\right)$$

  Where:

  - $A_t(\cdot)$: The **activation function** applied to transition t, such as ReLU, sigmoid, or tanh.
  - $\bullet t$: The set of all **input places** of transition t .
  - $W(p,t)$: The **weight** of the arc from place p to transition t.
  - $M(p)$: The current **marking** of place p.
  - $\sigma(t)$: The **bias term** associated with transition t.

- **Marking Update** $M'(p)$ **:** Once a transition t fires with activation output $\phi(t)$, the new marking $M'(p)$ for each place p is updated as follows:

$$M'(p) = M(p) + W(t,p) \cdot \phi(t) - W(p,t)$$

  Where:

  - $M'(p)$: The **updated marking** of place p after transition t fires.
  - $M(p)$: The **current marking** in place p before the transition fires.
  - $W(t,p)$: The **weight** of the arc from transition t to place p .
  - $\phi(t)$: The **activation output** of transition t..
  - $W(p,t)$: The **weight** of the arc from place p to transition t .

1.3.4 Deep Petri Nets Architecture

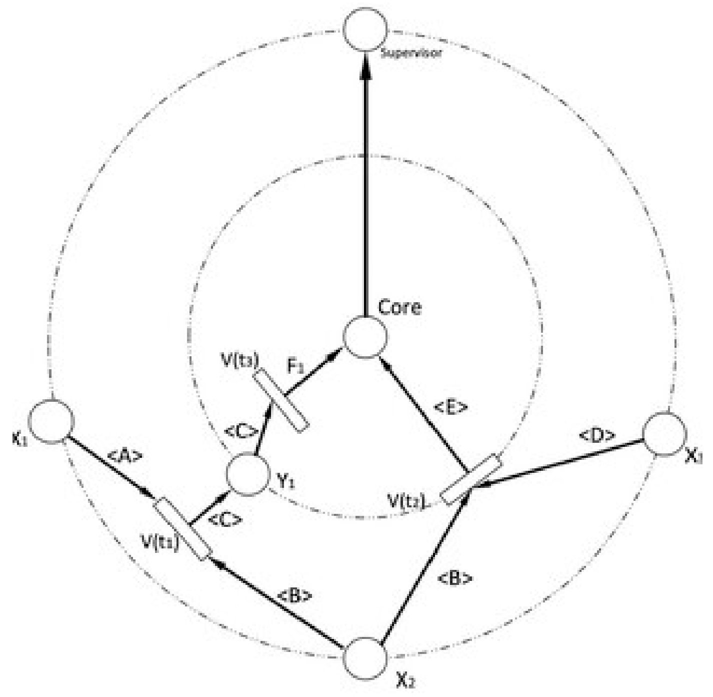


Figure 1.30: Deep Petri Net Architecture

Components of Deep Petri Net

- **Places**: Represent input variables or observed features.
- **Transition**: Embed neural networks to compute activation/decisions.
- **Rings**:
  - **Outer Ring**: Contains initial Places with tokens. These places have no incoming arcs.
  - **Intermediate Ring**: Contains Transitions that process tokens. Represents activation functions.
  - **Core Ring**: Contains final Places (those with no outgoing arcs). Represents the final results of the DPN execution.
- **Supervisor**: An external place with an arrow coming from the Core it may represent a global output, a controller, or a supervisory function. [31]

1.4 Model-Driven Architecture (MDA)

1.4.1 Definition

Model-Driven Architecture (MDA) is a software design approach defined by the Object Management Group (OMG) that promotes the use of models as the primary artifacts in the software development process. MDA separates the specification of system functionality from the specification of the implementation on a specific platform. [32]

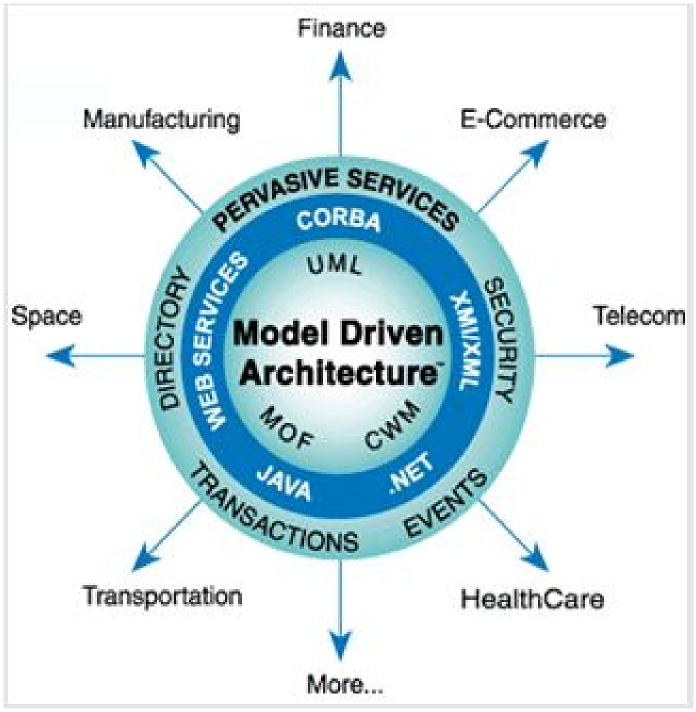


Figure 1.31: Model-Driven Architecture

1.4.2 Basic Concepts of MDA

- **System:** A system is a set of interrelated components working together toward a common goal. [33]
- **Model:** A model is a simplified representation of a system built to understand, analyze, or predict its behavior. [34]
- **Metamodel:** A metamodel defines the abstract syntax and semantic rules of a modeling language. It describes what elements can appear in a model and how they relate. [35]
- **Meta-metamodel:** A meta-metamodel defines the language used to build metamodels. It formalizes the most abstract modeling constructs and enables the definition of domain-specific modeling languages. [36]

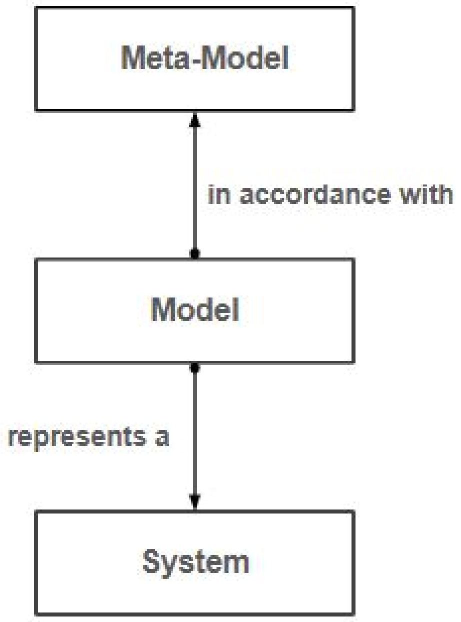


Figure 1.32: Relationship between system, model and meta-model

1.4.3 Architecture of MDA

Model-Driven Architecture (MDA) relies on a four-layer meta-modeling architecture, which provides a formal foundation for defining and managing models and their relationships.

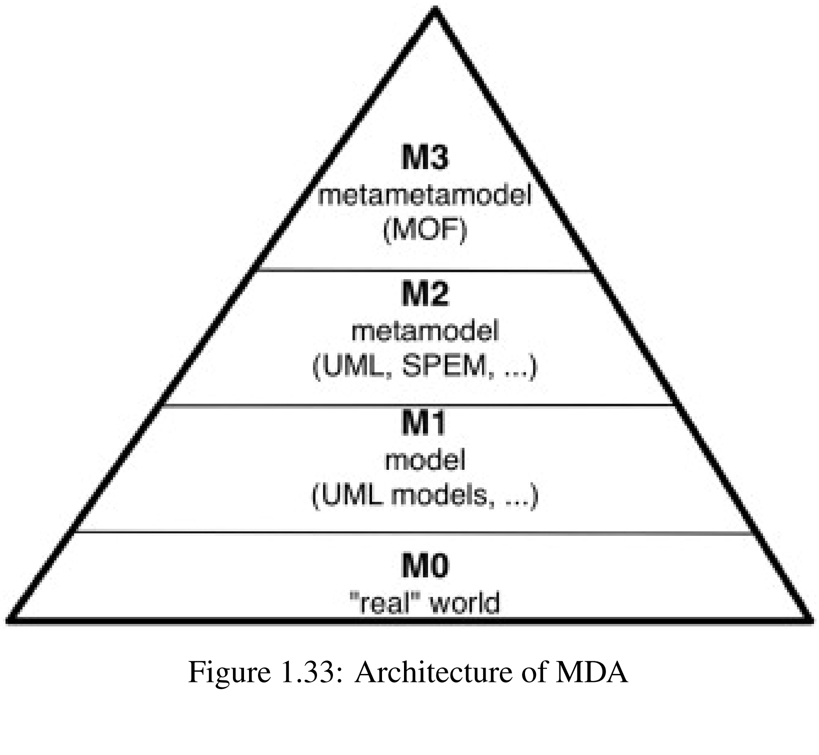


Figure 1.33: Architecture of MDA

The four layers are as follows [37]:

- **M0 Layer (Instance Layer):** This Layer corresponds to the real world. It includes the actual user data and real-world entities, which are instances of the models defined at the M1 Layer.
- **M1 Layer (Model Layer):** This Layer consists of information models that describe the data and behavior of real-world systems represented in M0. These models are instances of metamodels defined at the M2 Layer.
- **M2 Layer (Metamodel Layer):** This Layer defines the modeling languages and the grammar used to construct M1 models. Metamodels are instances of the meta-metamodel defined at M3.
- **M3 Layer (Meta-metamodel Layer):** This Layer consists of a single entity: the Meta-Object Facility (MOF). MOF defines the structure of metamodels, and provides mechanisms to extend or modify them. It is self-descriptive, meaning it can define its own structure.

1.4.4 Types of MDA

- **CIM:** The Computation Independent Model (CIM) corresponds to domain or business models that are entirely independent of any technical implementation. It captures the users' requirements using the terminology and concepts familiar to domain experts and practitioners.
- **PIM:** The Platform Independent Model (PIM) corresponds to the specification of the business logic of an application. It results from a software analysis aimed at satisfying business requirements without considering any specific implementation technology.
- **PSM:** The Platform Specific Model (PSM) corresponds to the specification of an application after it has been mapped onto a specific technological platform. It incorporates platform-specific details required for implementation. [38]

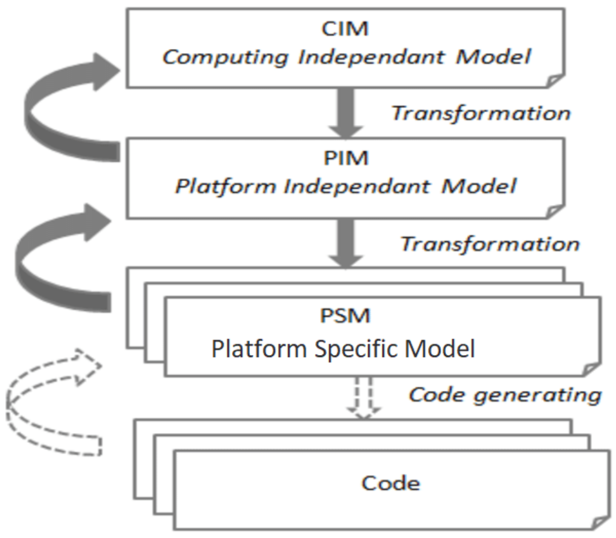


Figure 1.34: Types of MDA

1.5 Model Transformation

Model transformation is a fundamental aspect of Model-Driven Engineering (MDE). It consists of systematically converting a source model (Ma) into a target model (Mb) to make models executable or usable in downstream processes. This transformation can be endogenous or exogenous. Such transformations are essential for achieving objectives like code generation, model refactoring, and technology migration. [39]

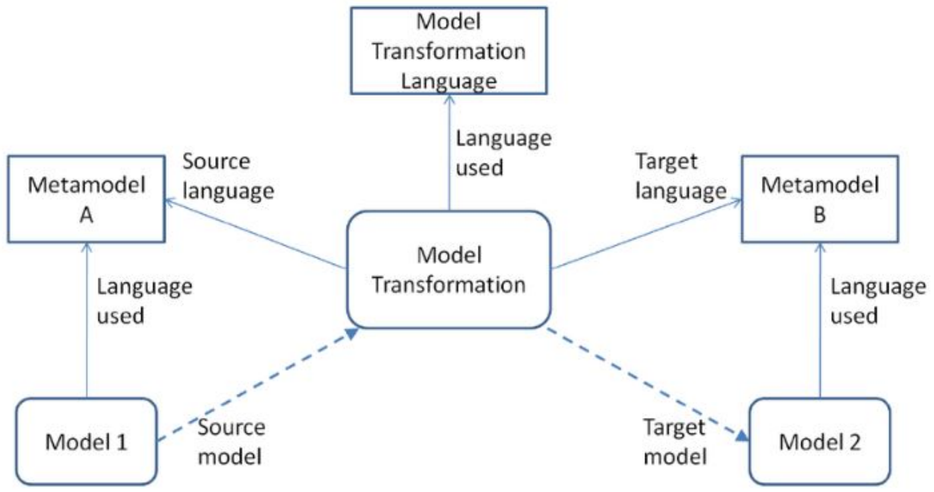


Figure 1.35: Model Transformation

1.5.1 Types of Model Transformation

- **Endogenous Transformation**: Both the source and target models conform to the same metamodel. [40]
- **Exogenous Transformation**: The source and target models conform to different metamodels. [40]
- **Horizontal Transformation**: The transformation occurs between models at the same level of abstraction. [41]
- **Vertical Transformation**: The transformation occurs between models at different levels of abstraction, typically refining a high-level model into a more detailed one. [41]

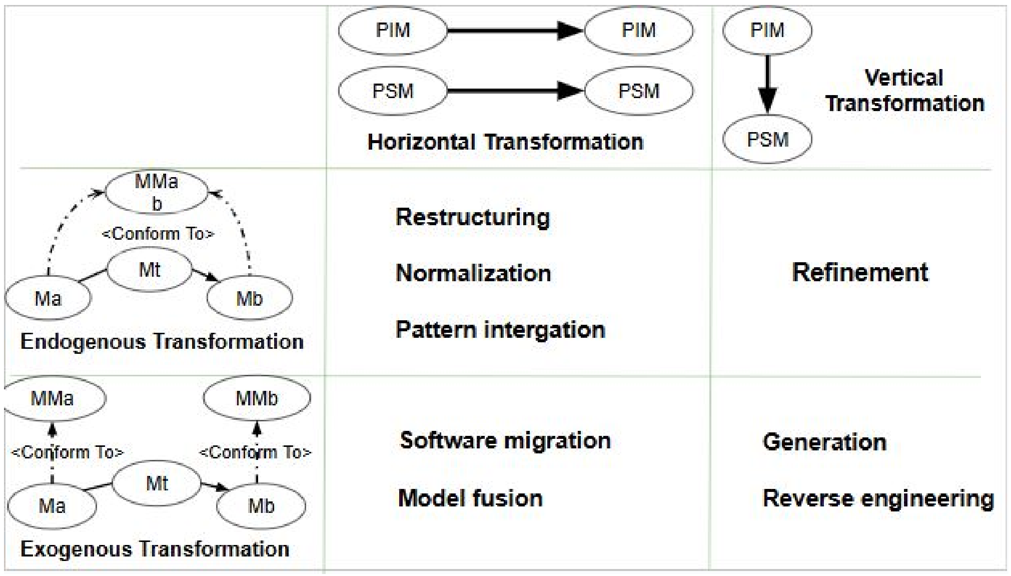


Figure 1.36: Types of model transformation

1.5.2 Classification of Model Transformation Approaches

Based on the classification, model transformations are typically divided into two main categories:

- **Model-to-Model Transformation (M2M)**:

  Refers to the process of automatically converting a source model into a target model, both of which conform to metamodels. It enables abstraction changes, model refinement, or transitions between design phases.

  The techniques for transformations of this type can be classified into five categories [43]:

  - Direct manipulation approaches.
  - Relational approaches.
  - Hybrid approaches.
  - Graph transformation-based approaches.
  - Structure-driven approaches.

- **Model-to-Code Transformation (M2C)**:

  Refers to the automatic generation of executable source code or configuration files from high-level models. This step bridges the gap between abstract design and implementation.

  In the model-to-code category, we distinguish between: [42]

  - **Visitor-Based Approaches**:

    These approaches traverse the model structure programmatically using the visitor design pattern. Custom code is written to visit each element in the model and generate the corresponding textual representation. Although flexible and powerful, visitor-based transformations require more effort and are typically harder to maintain.

- **Template-Based Approaches**:

  These approaches use predefined templates that define how model elements should be rendered into code. Tools like *Xpand*, *Acceleo*, and *Xtend* follow this paradigm. Template-based transformations are more readable, easier to maintain, and better suited for non-programmers or domain experts who wish to define generation rules declaratively.

Remark. In our work, we adopt a Model-to-Model (M2M) transformation approach to systematically convert models defined in one metamodel into corresponding models in another metamodel.

1.6 Graph Transformation

To understand graph transformation, we'll first go over the basics of graphs.

1.6.1 Basic Concepts of Graph:

A graph is a fundamental mathematical structure used to represent relationships between objects.
It is defined as a pair : [44]

$$\mathrm{G} = (N, A)$$

where:

- N is a finite set of **nodes** representing the objects.
- A is a set of **arcs** representing the relationships between pairs of nodes.

There are two types of graphs: undirected graphs (where nodes are connected by edges) , and directed graphs (where nodes are connected by arcs, which are edges with a direction).

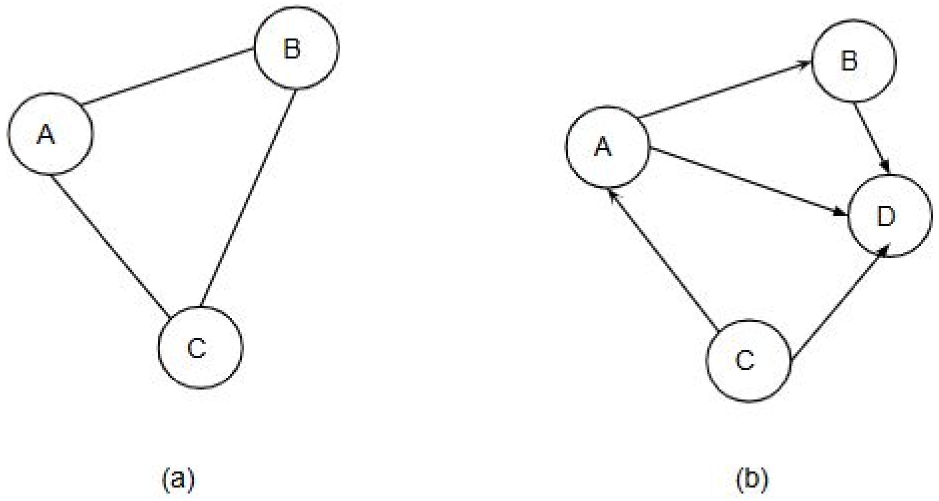


Figure 1.37: (a) undirected graph (b) directed graph

1.6.2 Graph Transformation Tools

There are several graph transformation tools such as: the Attributed Graph Grammar System (AGG), Triple Graph Grammar (TGG), A Tool for Multi-formalism and Meta-Modelling (AToMş), A Tool for Multi-Paradigm Modeling (AToMPM), and Henshin. [44]

In our work, we use " TGG " for its distinctive advantages: [45]

- **Bidirectional Transformations:** TGGs support both forward and backward transformations, enabling synchronization in both directions between source and target models.
- **Declarative and Rule-Based:** Transformations are specified through rules that declaratively define correspondences between models, making them easier to manage and maintain.
- **Automatic Generation of Synchronization Code:** From TGG rules, it is possible to automatically generate transformation engines that ensure consistency and support model synchronization.

1.7 Triple Graph Grammar (TGG)

Triple Graph Grammar (TGG) is a formal framework used in model-driven engineering to specify and manage bidirectional model transformations. It defines a set of correspondence rules that relate elements of a source model, a target model, and an intermediate correspondence model. TGGs enable the automatic synchronization and consistency maintenance between models, ensuring that changes in one model are reflected in the other. This makes TGG particularly suitable for round-trip engineering and model integration tasks. [46]

In our work, we use Eclipse Modeling Framework (EMF) to define th metamodels ,and Atlas Transformation Language (ATL) for the model to model transformation.

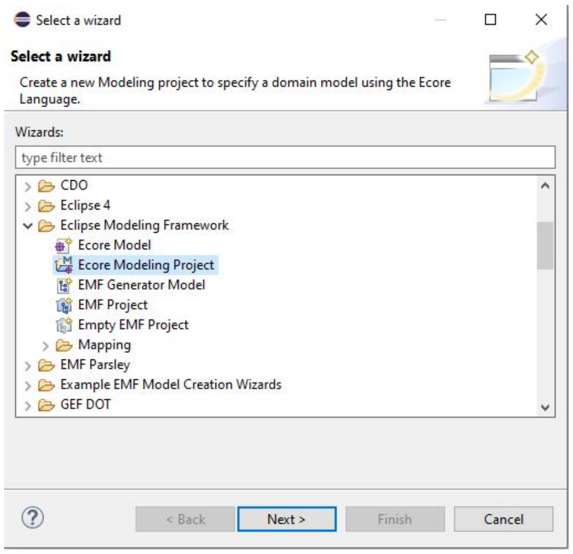
1.7.1 Eclipse Modeling Framework (EMF)

The EMF is a modeling framework and code generation facility provided by the Eclipse Foundation. It enables developers to define, manipulate, and generate structured data models in a *model-driven engineering* (MDE) context.

At its core, EMF allows users to define metamodels using a modeling language called **Ecore**, which is a subset of the OMG's *Meta Object Facility* (MOF) standard. These metamodels specify the structure of data in terms of: classes , attributes, references and inheritance. [47]

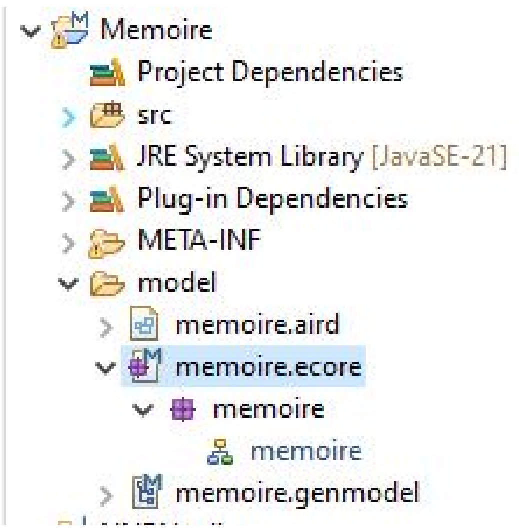
Once a metamodel is defined, EMF can automatically generate Java code that includes:

- Java interfaces and implementation classes for the model elements
- A factory class to instantiate the model
- A resource and serialization infrastructure (based on XML/XMI).

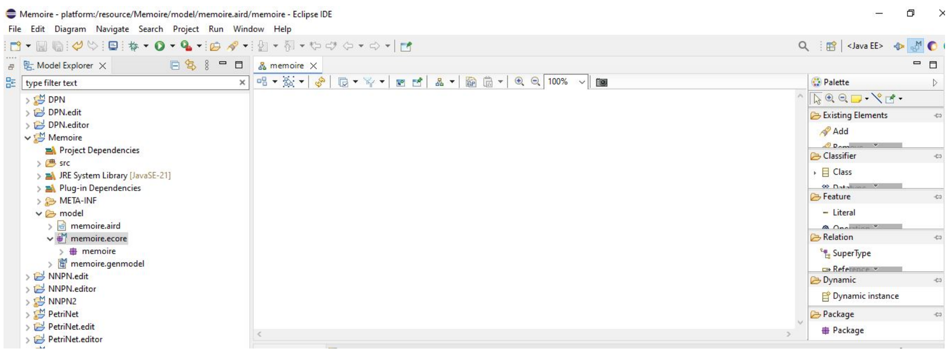

Select a wizard

Create a new Modeling project to specify a domain model using the Ecore Language.

(a) EMF Project



(b) Structure of the EMF



(c) EMF Editor

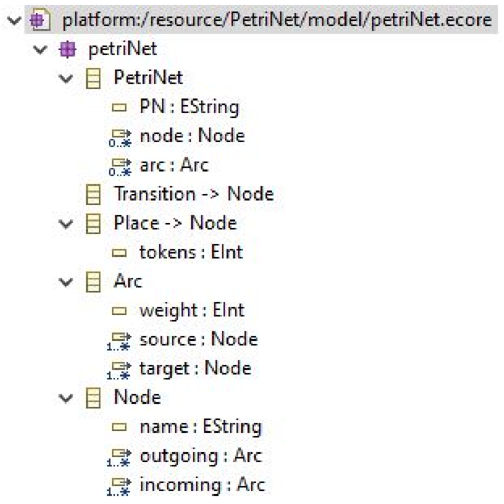
Figure 1.38: Presentation of the EMF Tool

1.7.2 Atlas Transformation Language (ATL)

Developed by the Atlas group at INRIA, ATL is specifically designed for Model-Driven Engineering (MDE) and enables model-to-model (M2M) transformations. ATL provides a way to define how elements from a source model are mapped and transformed into elements of a target model, both of which are typically defined using Ecore metamodels in the Eclipse Modeling Framework (EMF).

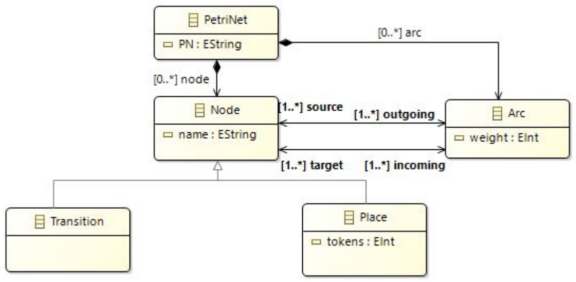
ATL supports both declarative and imperative paradigms:

- **Declarative rules**: Describe what elements should be matched and how they should be transformed.
- **Imperative sections**: Called *helpers* and *called rules*, they allow more control for complex logic that cannot be expressed purely declaratively. [49]

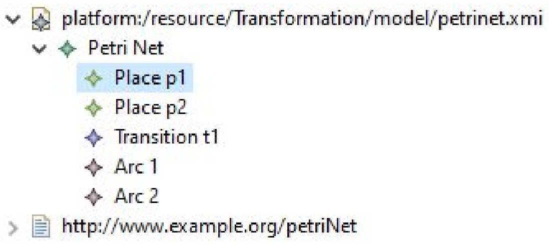


```
platform:/resource/PetriNet/model/petriNet.ecore
  petriNet
    PetriNet
      PN : EString
      node : Node
      arc : Arc
    Transition -> Node
    Place -> Node
      tokens : EInt
    Arc
      weight : EInt
      source : Node
      target : Node
    Node
      name : EString
      outgoing : Arc
      incoming : Arc
```

(a) Ecore File

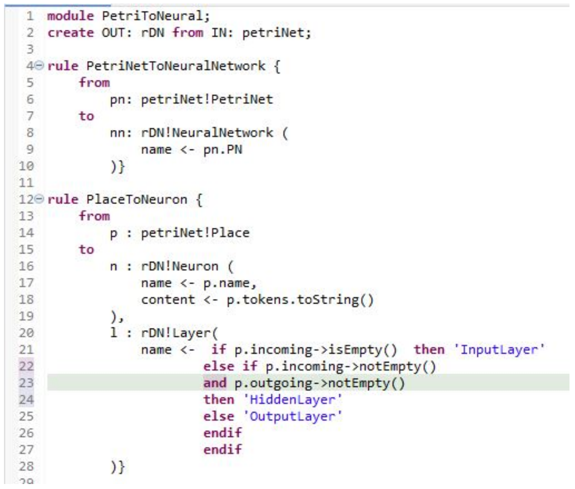


(b) PN Metamodel



```
platform:/resource/Transformation/model/petrinet.xmi
  Petri Net
    Place p1
    Place p2
    Transition t1
    Arc 1
    Arc 2
http://www.example.org/petriNet
```

(c) XMI File



```
module PetriToNeural;
create OUT: rDN from IN: petriNet;

rule PetriNetToNeuralNetwork {
    from
        pn: petriNet!PetriNet
    to
        nn: rDN!NeuralNetwork (
            name <- pn.PN
        )}

rule PlaceToNeuron {
    from
        p : petriNet!Place
    to
        n : rDN!Neuron (
            name <- p.name,
            content <- p.tokens.toString()
        ),
        l : rDN!Layer(
            name <-  if p.incoming->isEmpty()  then 'InputLayer'
                  else if p.incoming->notEmpty()
                  and p.outgoing->notEmpty()
                  then 'HiddenLayer'
                  else 'OutputLayer'
                  endif
                  endif
        )}
```

(d) ATL File

Figure 1.39: Presentation of the ATL tool

Conclusion

This chapter presented the integration of Petri Nets and neural networks through Deep Petri Nets (DPNs), combining formal modeling with adaptive learning. We also introduced key model transformation concepts within the Model-Driven Architecture (MDA), highlighting the use of Triple Graph Grammars (TGGs), EMF, and ATL to automate and synchronize hybrid model generation. These foundations enable the development of intelligent systems that are both structured and adaptive.

Chapter2

Related Works

Introduction

In this chapter, we review existing works related to the modeling and simulation of complex systems using Petri Nets, neural networks, and hybrid models. We also discuss model transformation approaches that are relevant to our methodology. This review allows us to identify research gaps and position our contribution within the state of the art.

2.1 Petri Nets in System Modeling

Petri Nets are a well-established formalism for modeling and analyzing discrete event systems, especially those characterized by concurrency, synchronization, and resource sharing. [57]

Over the decades, Petri Nets have been widely applied in various domains such as workflow management systems, distributed computing, manufacturing systems, communication protocols, and biological systems modeling [**?**, 1]. Their ability to capture both control flow and data flow makes them particularly suitable for systems that involve parallelism and synchronization.

Many extensions have been developed to enhance their expressiveness, including Colored Petri Nets (CPNs) [58], which allow tokens to carry data; Timed Petri Nets, which model temporal behavior; and Stochastic Petri Nets [59], which incorporate probabilistic aspects. These variants further broaden the applicability of Petri Nets in real-world systems that require richer semantic modeling.

The strength of Petri Nets lies in their formal analysis capabilities, including reachability, deadlock detection, liveness, and boundedness [1]. These properties make them valuable not only for system design but also for formal verification and validation.

Despite their robustness in structure and behavior representation, classical Petri Nets face limitations when applied to systems that require learning, adaptation, or handling uncertain and fuzzy information. These challenges have led to research efforts focused on combining Petri Nets with artificial intelligence techniques, especially neural networks, to create hybrid intelligent modelsan area discussed in the following sections.

2.2 Neural Networks and Learning-based Models

With the rise of computational power and large datasets, deep learning has emerged as a powerful subfield of machine learning. Deep Neural Networks (DNNs), particularly architectures like Convolutional Neural Networks (CNNs) [60] and Recurrent Neural Networks (RNNs) [61], have achieved remarkable success in fields such as image recognition, natural language processing, and autonomous systems.

Deep learning models are able to automatically extract hierarchical representations from data, which makes them highly effective in handling unstructured and high-dimensional information. However, these models are often seen as "black-box" systems due to their lack of transparency and interpretability [62].

Despite their impressive performance, neural networks face challenges when used in applications that require formal guarantees, logical reasoning, or interpretability. This has led to increasing interest in combining neural networks with symbolic models such as Petri Nets, aiming to leverage the strengths of both paradigms.

2.3 Hybrid Models: Integration of Petri Nets and Neural Networks

Given the limitations of both classical Petri Nets in learning and adaptation, and neural networks in explainability and formal verification, hybrid models have emerged to leverage the strengths of both paradigms. These models aim to combine the structured, formal nature of Petri Nets with the data-driven, adaptive capabilities of neural networks, forming a new class of intelligent systems suitable for modeling complex and dynamic behaviors.

2.3.1 Fuzzy Petri Nets

Fuzzy Petri Nets (FPNs) extend classical Petri Nets by integrating fuzzy logic to deal with imprecision and uncertainty. In these models, transitions are associated with fuzzy rules, and tokens can carry fuzzy values, allowing for approximate reasoning [63]. FPNs have been widely used in expert systems, risk analysis, and decision support systems.

More recently, researchers have proposed combining fuzzy logic with neural networks within the Petri Net framework. For example, Liu et al. [64] introduced the Deep Fuzzy Petri Net (DFPN), a hybrid framework that enables explainable decision-making by integrating fuzzy rule bases with deep learning mechanisms.

2.3.2 Neural Petri Nets

Neural Petri Nets (NPNs) are models in which the structure of the Petri Net is embedded with neural components, such as weights and activation functions. The flow of tokens can be influenced by neural computations, enabling the Petri Net to adapt its behavior based on learning processes [65]. These models are particularly suited for applications in dynamic environments and real-time decision systems.

2.3.3 Deep Petri Nets

Deep Petri Nets (DPNs) represent a more recent and advanced form of integration, where deep learning models are embedded into Petri Net structures.

DPNs have been applied to various domains including industrial automation, intelligent monitoring systems, and autonomous decision-making. Their dual nature offers a promising path toward building systems that are both formally verifiable and capable of adaptive learning.

2.4 Comparative Analysis of Hybrid Petri Net Approaches

To highlight the effectiveness of our proposed approaches, we present a comparative analysis with recent hybrid models that integrate Petri nets and learning-based techniques. The table below summarizes various methods in terms of their integration strategies, main features, and limitations. This comparison underlines how our work offers a more structured, automated, and traceable solution.

Approach	**Integration Method**	**Main Features**	**Limitations / Remarks**
Liu et al. (Deep Fuzzy Petri Nets) [50]	Fuzzy Logic + Deep Learning	Captures uncertainty and non-linear behavior; interpretable	Requires expert-defined rules and parameters
Zhao et al. (Fuzzy Petri Nets) [51]	Fuzzy weights + Probabilistic transitions	Better handles imprecision in dynamic environments	Limited automation and traceability
Wang and Zhao (Expert Systems) [52]	Fuzzy Petri Net with adaptive learning	Supports dynamic rule evaluation and real-time decisions	Focused on expert systems; lacks structural automation
Kordon et al. (MDE with Petri + ML) [53]	Metamodeling + MDE integration	Formal structure using MDE; structural modeling with AI	Integration complexity; limited to structure
Zhang et al. (Deep Reinforcement PN) [54]	RL agents within Petri transitions	Combines learning and Petri for autonomous systems	Less focus on formal traceability and reuse
Yin et al. (PN-based NN interpretation) [55]	PN transitions mimic neuron activations	Improves explainability in deep models	Targets interpretation, not transformation
Zhao et al. (DRL for scheduling) [56]	DRL within PN structure	Real-time optimization in manufacturing systems	Domain-specific, limited reusability
Our Approaches	Metamodeling + ATL + TGG (two approaches)	Automatic transformation; clear modular structure; traceability; formal verification support	More complex setup; requires metamodel expertise

Table 2.1: Comparative Analysis of Our Approaches

As shown in Table 2.1, several hybrid approaches have explored the integration of Petri nets with fuzzy logic, machine learning, or deep reinforcement learning. For instance, works like Liu et al. [50] and Zhao et al. [51] emphasize handling uncertainty and non-linear behaviors through fuzzy rules and probabilistic transitions. However, these models often depend on manually defined parameters and lack formal traceability.

Other approaches such as those by Kordon et al. [53] and Zhang et al. [54] attempt to integrate learning mechanisms within Petri Net structures using MDE or reinforcement learning techniques. While they offer more automation and adaptive behavior, they are often limited in reusability or lack a standardized transformation process.

Our approach distinguishes itself by adopting a model-driven engineering (MDE) strategy, using ATL and TGG to ensure a formal and automated transformation from classical Petri Nets to Deep Petri Nets. It ensures structural clarity, traceability, and formal verification supportadvantages not fully addressed in the compared models. Although our setup requires expertise in metamodeling, it provides a reusable and extensible framework applicable across domains.

2.5 Conclusion

This chapter reviewed foundational and recent works on Petri Nets, neural networks, and their hybridization. We discussed the strengths and limitations of symbolic and subsymbolic models, as well as attempts to integrate them using fuzzy logic, neural components, and deep learning techniques.

While previous studies provide valuable insights and technical contributions, they often lack generality, traceability, or formal integration methodologies. In response to these limitations, our work proposes a model-driven transformation approach based on ATL and EMF to bridge the gap between symbolic Petri Nets and adaptive neural architectures.

The next chapter presents our proposed framework and transformation strategies in detail.

Contributions

Chapter3

Proposed Approaches

Introduction

This chapter introduces two model transformation approaches for converting a classical Petri Net into a Deep Petri Net (DPN). The main objective is to combine the formal semantics of Petri Nets with the adaptive learning capabilities of neural networks, enabling more advanced modeling and simulation of complex systems. These transformations are implemented using the Atlas Transformation Language (ATL) and are based on well-defined metamodels within the Eclipse Modeling Framework (EMF).

The first approach follows a two-step process: initially, the Petri Net is transformed into a Neural Network model, which serves as an intermediate representation. This intermediate model is then further transformed into a Deep Petri Net. The second approach involves integrating elements of the Petri Net and the Neural Network directly into a unified model, which is subsequently transformed into a Deep Petri Net.

This chapter also details the metamodels used, the transformation rules, and the ATL implementation of each step, demonstrating the feasibility and effectiveness of transitioning from formal models to intelligent, learning-capable systems.

3.1 The First Approach:

In this approach, we first transform the Petri Net model into a Neural Network model. This initial transformation aims to reinterpret the structural and behavioral elements of the Petri Net in terms of neural components. Once the resulting is obtained, it is then transformed into a Deep Petri Net model.

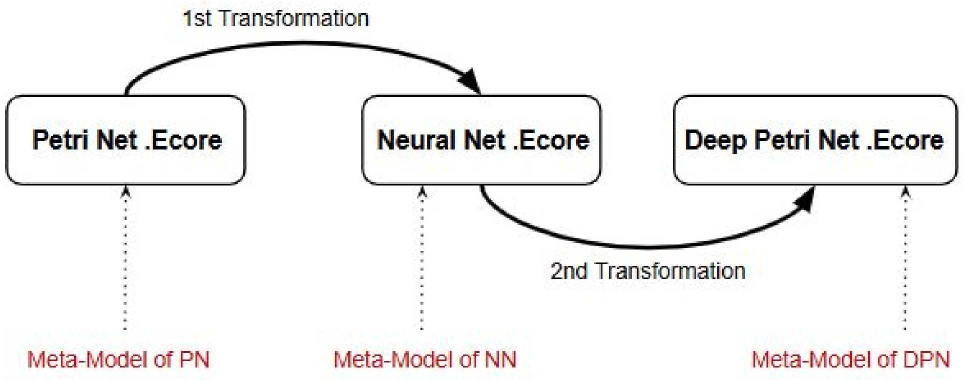


Figure 3.1: Architecture of First Approach

3.1.1 Meta Model The Petri Net (PN)

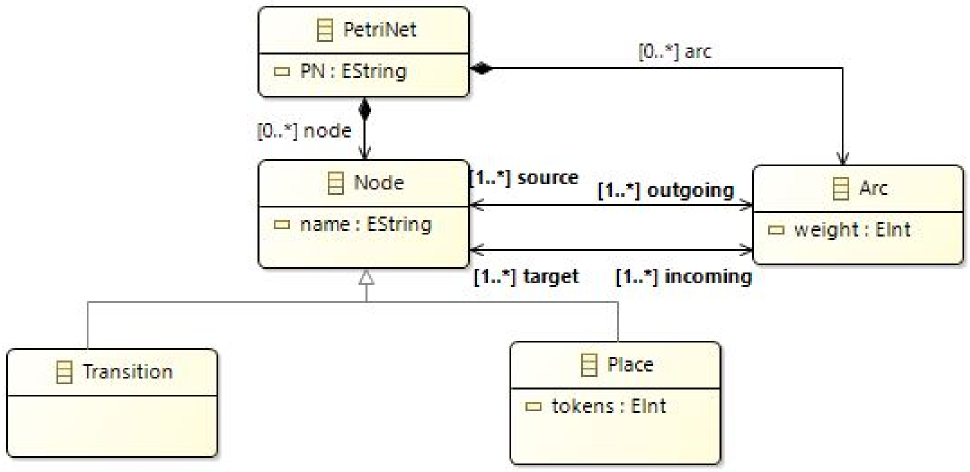
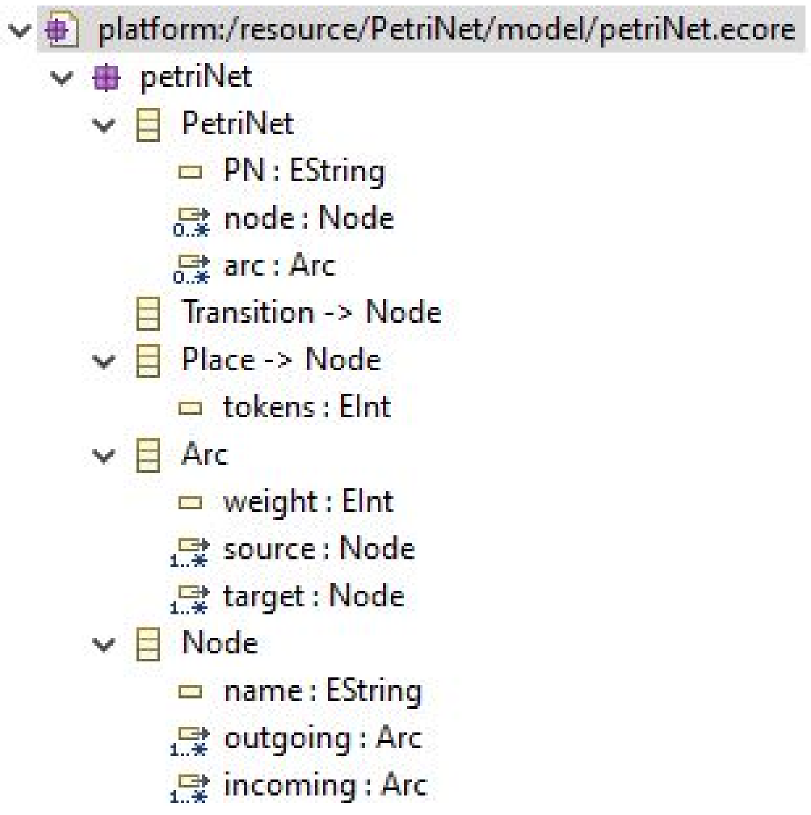


Figure 3.2: Meta Model of PN

The metamodel is manually created using the EMF graphical editor. Saving the model automatically generates the .ecore file.



```
platform:/resource/PetriNet/model/petriNet.ecore
  petriNet
    PetriNet
      PN : EString
      node : Node
      arc : Arc
    Transition -> Node
    Place -> Node
      tokens : EInt
    Arc
      weight : EInt
      source : Node
      target : Node
    Node
      name : EString
      outgoing : Arc
      incoming : Arc
```

Figure 3.3: PetriNet.ecore Elements

The provided metamodel defines the structure of a Petri Net. It outlines the main components and the relationships between them

- **Class `PetriNet`**:
  This is the main container class representing an entire Petri Net model. It includes:
  - A `name` attribute (`PN`).
  - A collection of `Node` elements.
  - A collection of `Arc` elements.
- **Abstract Class `Node`**:
  This is a general superclass for all elements that can appear in a Petri Net. It has:
  - A `name` attribute of type `EString`.
  - Two concrete subclasses: `Place` and `Transition`.
- **Class `Place`** (subclass of `Node`):
  Represents a location capable of holding tokens. It contains:
  - A `tokens` attribute of type `EInt`.
- **Class `Transition`** (subclass of `Node`):
  Represents an event or activity that may occur, changing the state of the Petri Net by moving tokens between places.
- **Class `Arc`**:
  Represents a directed connection between nodes, describing the flow of tokens. Each `Arc` includes:
  - A `weight` attribute (type `EInt`) to define the number of tokens transferred.
  - An association with a `source` node.
  - An association with a `target` node.

Relationships:

- An `Arc` always has a `source` and a `target` node.
- A `Node` can have one or more outgoing and incoming `Arc`s.

3.1.2 Meta Model The Neural Network (NN)

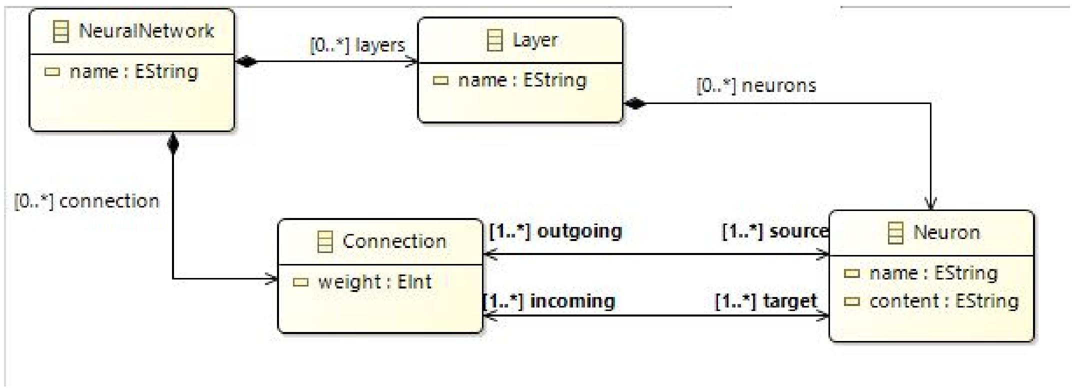
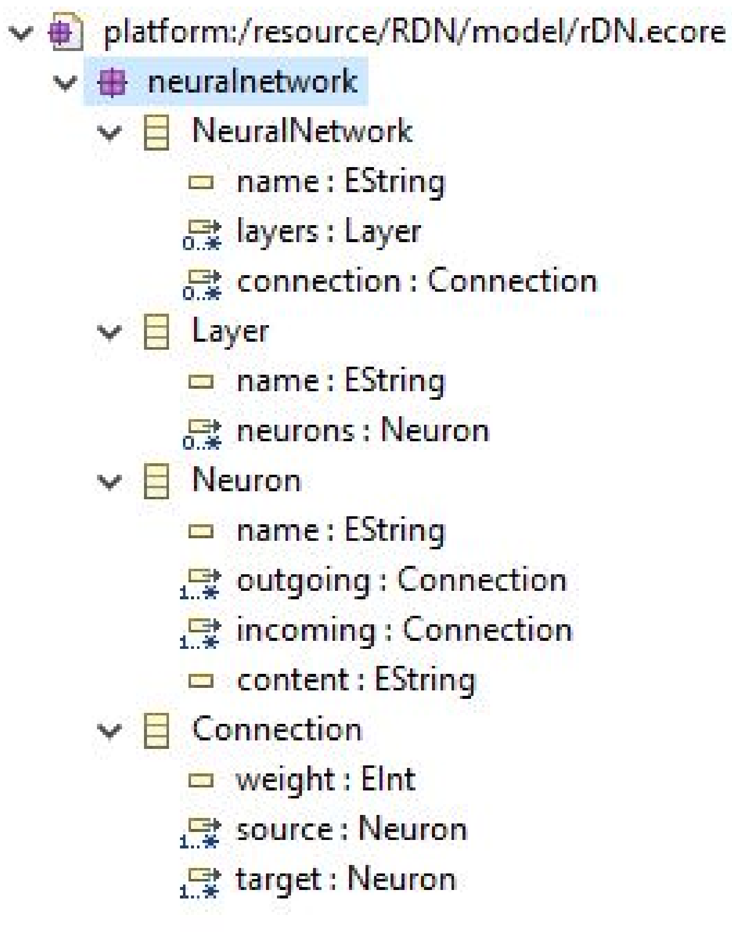


Figure 3.4: Meta Model of NN



```
platform:/resource/RDN/model/rDN.ecore
  neuralnetwork
    NeuralNetwork
      name : EString
      layers : Layer
      connection : Connection
    Layer
      name : EString
      neurons : Neuron
    Neuron
      name : EString
      outgoing : Connection
      incoming : Connection
      content : EString
    Connection
      weight : EInt
      source : Neuron
      target : Neuron
```

Figure 3.5: NeuralNetwork.ecore Elements

This metamodel represents the structural definition of a Neural Network .

- **Class `NeuralNetwork`**:
  This is the main container class that represents an entire neural network model. It includes:
  - A `name` attribute of type `EString`, used to identify the network.
  - A collection of `Layer` elements.
  - A collection of `Arc` elements.
- **Class `Layer`**:
  Represents a level in the neural network (input layer, hidden layers, output layer). Each `Layer` contains:
  - A `name` attribute of type `EString`.
  - A collection of `Neuron` elements
- **Class `Neuron`**:
  Each `Neuron` has:
  - A `name` attribute of type `EString`.
  - A `content` attribute of type `EString`, which may represent internal data such as the activation function.
- **Class `Connection`**:
  Represents a directed link between neurons that facilitates the flow of information. Each `Connection` includes:
  - A `weight` attribute of type `EInt`, representing the strength of the connection.
  - A `source` neuron.
  - A `target` neuron .

Relationships:

- Each `Neuron` can have one or more incoming and outgoing `Connections`.
- `Connections` link neurons across layers or within the same layer, depending on the network topology.

3.1.3 First Transformation

Once the metamodels are defined, we create the ATL files to specify the transformation rules .

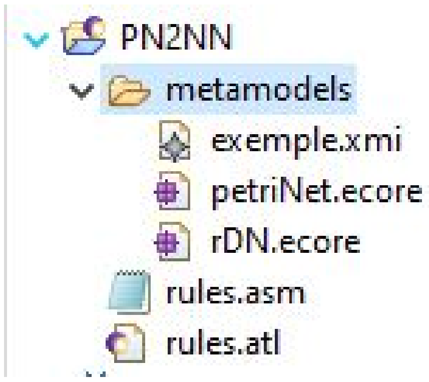
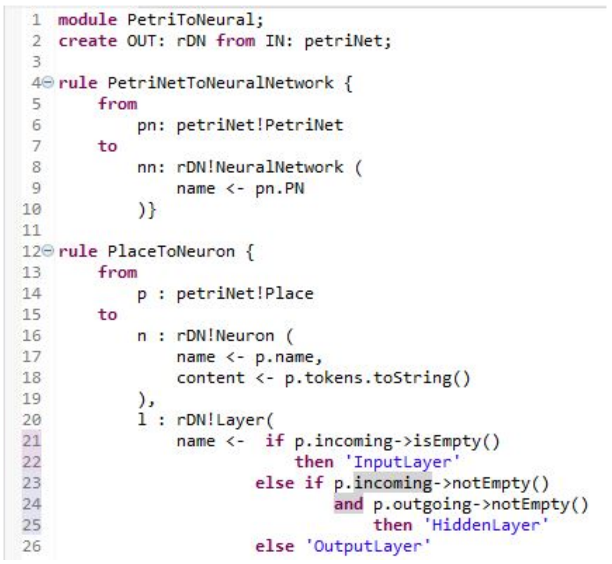


Figure 3.6: ATL Project

The ATL (Atlas Transformation Language) code shown in Figure 3.7 defines the transformation rules for converting a Petri Net model into a Neural Network model.

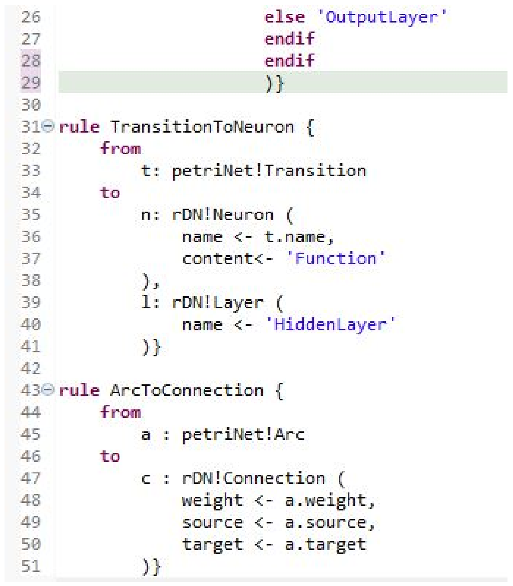


```
module PetriToNeural;
create OUT: rDN from IN: petriNet;

rule PetriNetToNeuralNetwork {
    from
        pn: petriNet!PetriNet
    to
        nn: rDN!NeuralNetwork (
            name <- pn.PN
        )}

rule PlaceToNeuron {
    from
        p : petriNet!Place
    to
        n : rDN!Neuron (
            name <- p.name,
            content <- p.tokens.toString()
        ),
        l : rDN!Layer(
            name <-  if p.incoming->isEmpty()
                        then 'InputLayer'
                    else if p.incoming->notEmpty()
                            and p.outgoing->notEmpty()
                                then 'HiddenLayer'
                    else 'OutputLayer'
```

(a)



```
                    else 'OutputLayer'
                    endif
                    endif
                    )}

rule TransitionToNeuron {
    from
        t: petriNet!Transition
    to
        n: rDN!Neuron (
            name <- t.name,
            content<- 'Function'
        ),
        l: rDN!Layer (
            name <- 'HiddenLayer'
        )}

rule ArcToConnection {
    from
        a : petriNet!Arc
    to
        c : rDN!Connection (
            weight <- a.weight,
            source <- a.source,
            target <- a.target
        )}
```

(b)

Figure 3.7: ATL transformation rules from Petri Net to Neural Network

Description of the ATL Rules:

- **Rule 1: `PetrinetToNeuralnetwork`**
  This rule transforms the entire Petri Net into a Neural Network . It simply copies the name attribute from the source model to the target model.

- **Rule 2: `PlaceToNeuron`**
  This rule converts each `Place` in the Petri Net into a `Neuron`. The neuron's name is set to the name of the place, and its content is initialized with the token count. The neuron is then assigned to a specific type of layer based on its arc connections:
  - No incoming arcs → assigned to the `InputLayer`
  - Both incoming and outgoing arcs → assigned to the `HiddenLayer`
  - Only incoming arcs → assigned to the `OutputLayer`
- **Rule 3: `TransitionToNeuron`**
  This rule transforms a `Transition` into a `Neuron`. The neuron's content is fixed as `"Function"` to represent processing logic, and it is always assigned to the `HiddenLayer`.
- **Rule 4: `ArcToConnection`**
  This rule transforms each `Arc` in the Petri Net into a `Connection` in the neural network. It preserves the weight of the arc and maps the source and target elements directly to the corresponding neurons.

Remark. The outcome of the transformation from the Petri net to the neural network model provides strong support for the validity of the metamodel used in "The Second Approach", known as the *Neural Network of Petri Net (NNPN)*. This intermediate metamodel effectively captures and unifies the structural and behavioral characteristics of both classical Petri nets and neural networks. By analyzing the transformation results, we observe clear alignment between elements of the two original formalisms, which demonstrates that NNPN can serve as a coherent and consistent foundation for integrating learning capabilities within a formal Petri net framework.

3.1.4 Meta Model The Deep Petri Net (DPN)

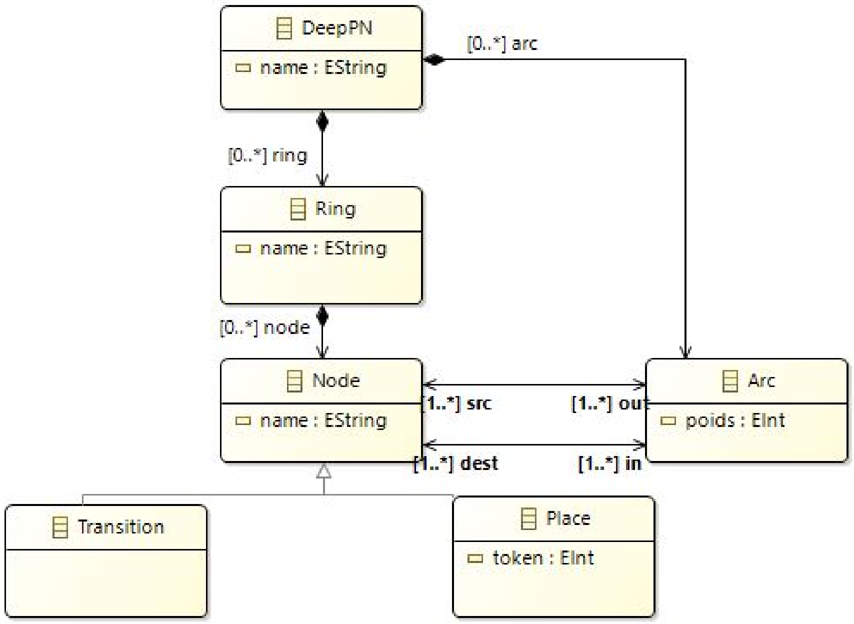
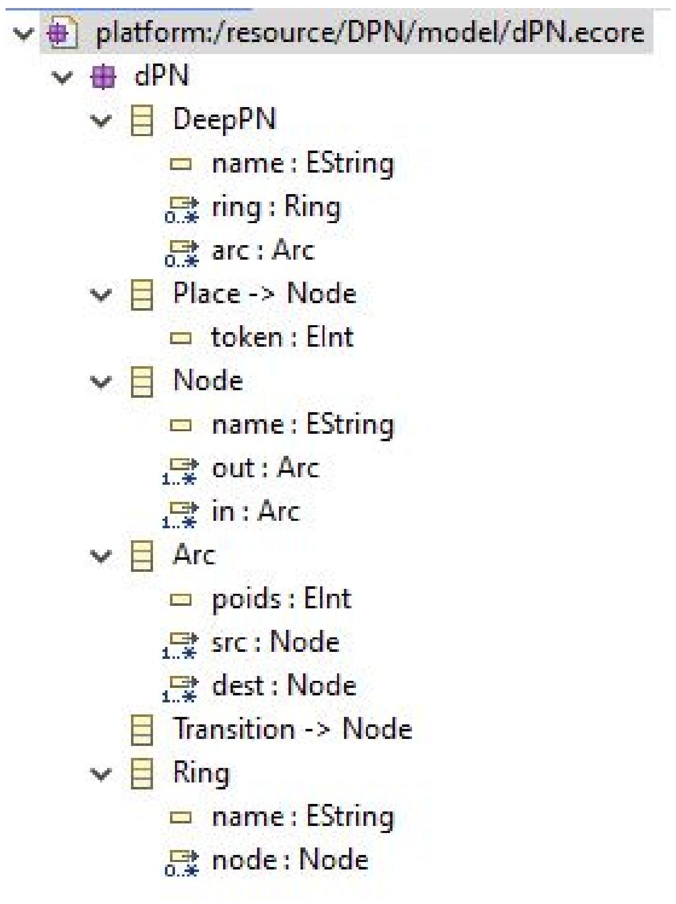


Figure 3.8: Meta Model of DPN



```
platform:/resource/DPN/model/dPN.ecore
  dPN
    DeepPN
      name : EString
      0..* ring : Ring
      0..* arc : Arc
    Place -> Node
      token : EInt
    Node
      name : EString
      1..* out : Arc
      1..* in : Arc
    Arc
      poids : EInt
      1..* src : Node
      1..* dest : Node
    Transition -> Node
    Ring
      name : EString
      0..* node : Node
```

Figure 3.9: DeepPetriNet.ecore Elements

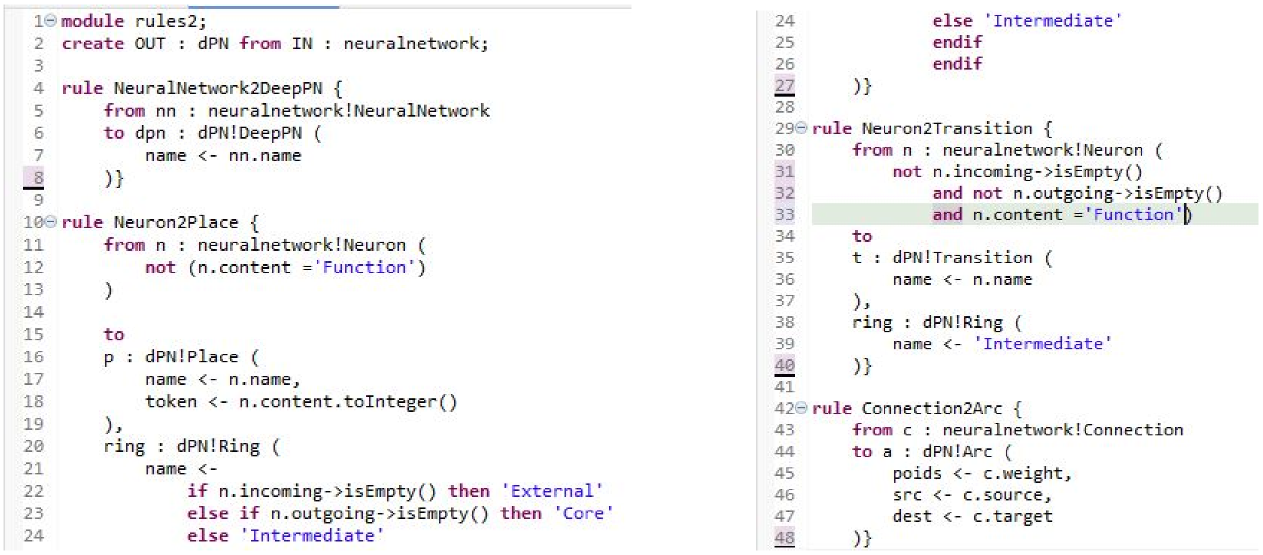
This metamodel defines the structure of a Deep Petri Net, an advanced formalism that integrates Petri Nets with layered

- **Class `DeepPN`**:
  The central class representing a Deep Petri Net. It includes:
  - A `name` attribute of type `EString`.
  - A collection of `Ring` elements.
  - A collection of `Arc` elements.
- **Class `Ring`**:
  Each `Ring` represents a structural layer in the DeepPN. It contains:
  - A `name` attribute of type `EString`.
  - A collection of `Node` elements (`Place` or `Transition`).

- **Class `Node`** (abstract):
  An abstract superclass representing elements of the net. Each `Node` has:
  - A `name` attribute of type `EString`.
  - One or more outgoing arcs (`out`).
  - One or more incoming arcs (`in`).
- **Class `Place`** (inherits from `Node`):
  Represents a state or condition within the system. It includes:
  - A `token` attribute of type `EInt`.
- **Class `Transition`** (inherits from `Node`):
  Represents an event or action.
- **Class `Arc`**:
  Models the connection between two `Node`s. It includes:
  - A `weight` attribute of type `EInt`, indicating the number of tokens involved.
  - A `src` (source node) and `dest` (target node) reference.

3.1.5 Second Transformation

Following the initial transformation -from a Petri Net to a Neural Network- , the resulting model is then used as input for a subsequent transformation into a Deep Petri Net.



```
module rules2;
create OUT : dPN from IN : neuralnetwork;

rule NeuralNetwork2DeepPN {
    from nn : neuralnetwork!NeuralNetwork
    to dpn : dPN!DeepPN (
        name <- nn.name
    )}

rule Neuron2Place {
    from n : neuralnetwork!Neuron (
        not (n.content ='Function')
    )

    to
    p : dPN!Place (
        name <- n.name,
        token <- n.content.toInteger()
    ),
    ring : dPN!Ring (
        name <-
            if n.incoming->isEmpty() then 'External'
            else if n.outgoing->isEmpty() then 'Core'
            else 'Intermediate'
```

(a)

```
            else 'Intermediate'
            endif
            endif
    )}

rule Neuron2Transition {
    from n : neuralnetwork!Neuron (
        not n.incoming->isEmpty()
            and not n.outgoing->isEmpty()
            and n.content ='Function')
    to
    t : dPN!Transition (
        name <- n.name
    ),
    ring : dPN!Ring (
        name <- 'Intermediate'
    )}

rule Connection2Arc {
    from c : neuralnetwork!Connection
    to a : dPN!Arc (
        poids <- c.weight,
        src <- c.source,
        dest <- c.target
    )}
```

(b)

Figure 3.10: ATL transformation rules from The resulting to Deep Petri Net

Description of the ATL Rules:

- **Rule 1: `NeuralNetwork2DeepPN`**
  This rule transforms the entire `NeuralNetwork` into a `DeepPN`. The name attribute is directly copied from the source model.

- **Rule 2: `Neuron2Place`**
  This rule converts a `Neuron` into a `Place` only if the neuron is not a function . It transfers:
  - the neuron's `name` and `content` to the `Place`,
  - classifies the resulting `Place` into an `Ring`:
    - ∗ No incoming connections → assigned to `External`,
    - ∗ No outgoing connections → assigned to `Core`,
    - ∗ Both input and output connections → assigned to `Intermediate`.
- **Rule 3: `Neuron2Transition`**
  This rule transforms a `Neuron` into a `Transition` and placed in the `Intermediate` if the following conditions are met:
  - It has both incoming and outgoing connections,
  - It represents a function,
- **Rule 4: `Connection2Arc`**
  This rule converts a `Connection` from the neural network into an `Arc` in the Deep Petri Net. The transformation includes:
  - copying the `weight` as `poids`,
  - and mapping the `source` and `target` neurons to their corresponding nodes in the DPN.

3.2 The Second Approach:

This approach involves integrating the Petri Net and Neural Network into a single model, which is then transformed into a Deep Petri Net.

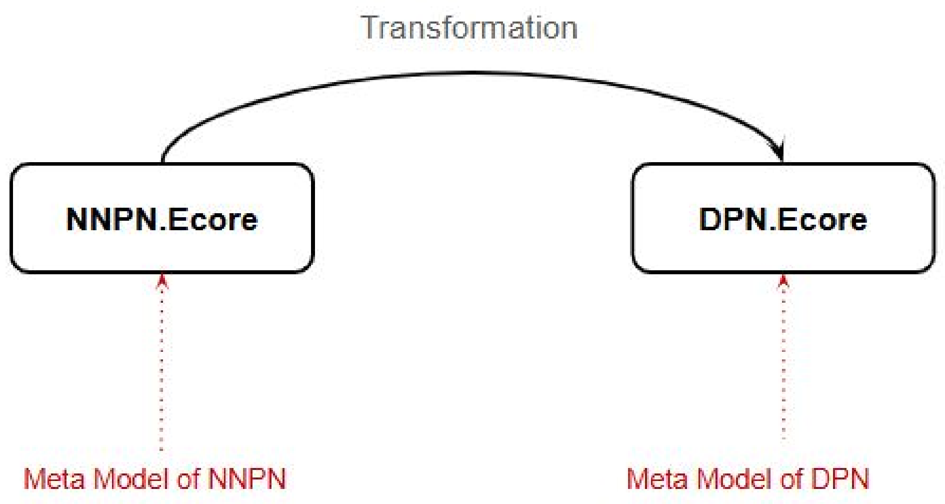


Figure 3.11: Architecture of Second Approach

3.2.1 Meta Model The Neural Network of Petri Net (NNPN)

This metamodel defines a unified structure that integrates elements of both Petri nets and neu-ral networks to support the modeling of systems with adaptive and analytical capabilities.

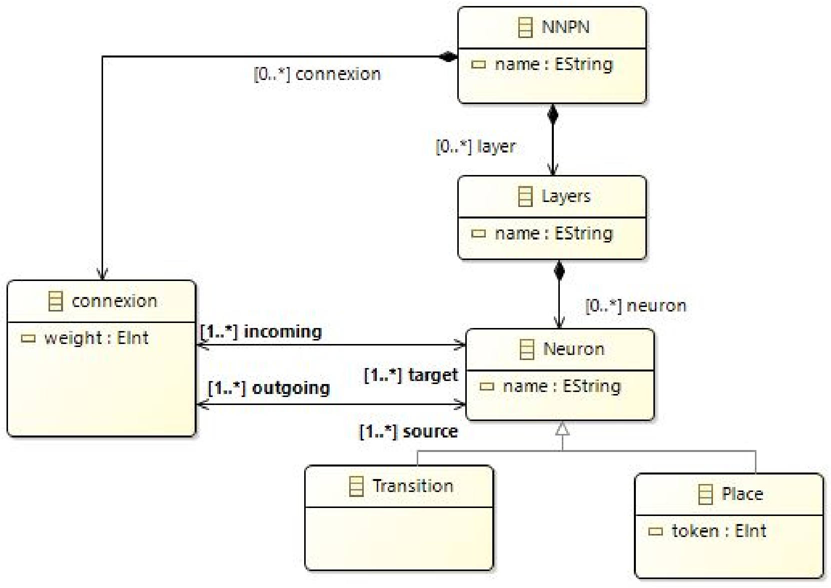
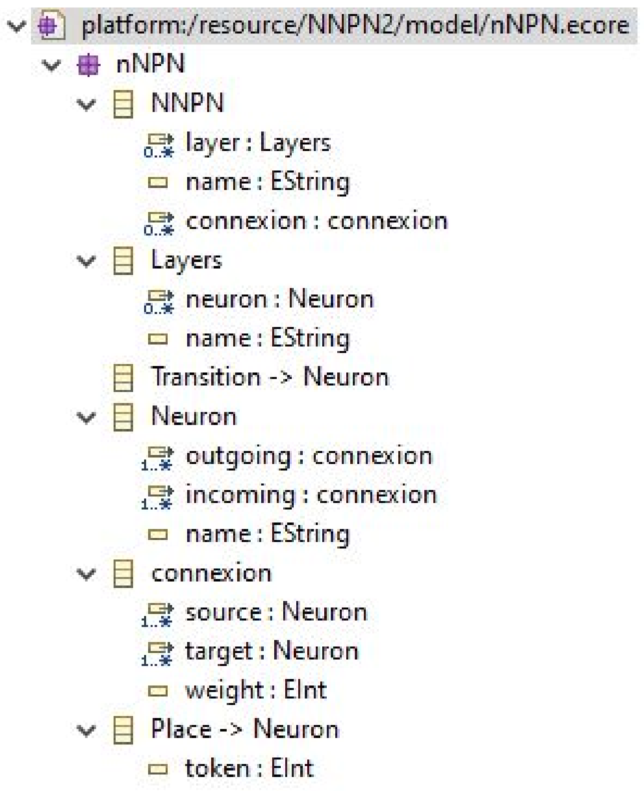


Figure 3.12: Meta Model of NNPN



platform:/resource/NNPN2/model/nNPN.ecore
- nNPN
  - NNPN
    - layer : Layers
    - name : EString
    - connexion : connexion
  - Layers
    - neuron : Neuron
    - name : EString
  - Transition -> Neuron
  - Neuron
    - outgoing : connexion
    - incoming : connexion
    - name : EString
  - connexion
    - source : Neuron
    - target : Neuron
    - weight : EInt
  - Place -> Neuron
    - token : EInt

Figure 3.13: NNPN.ecore Elements

It is composed of the following core components:

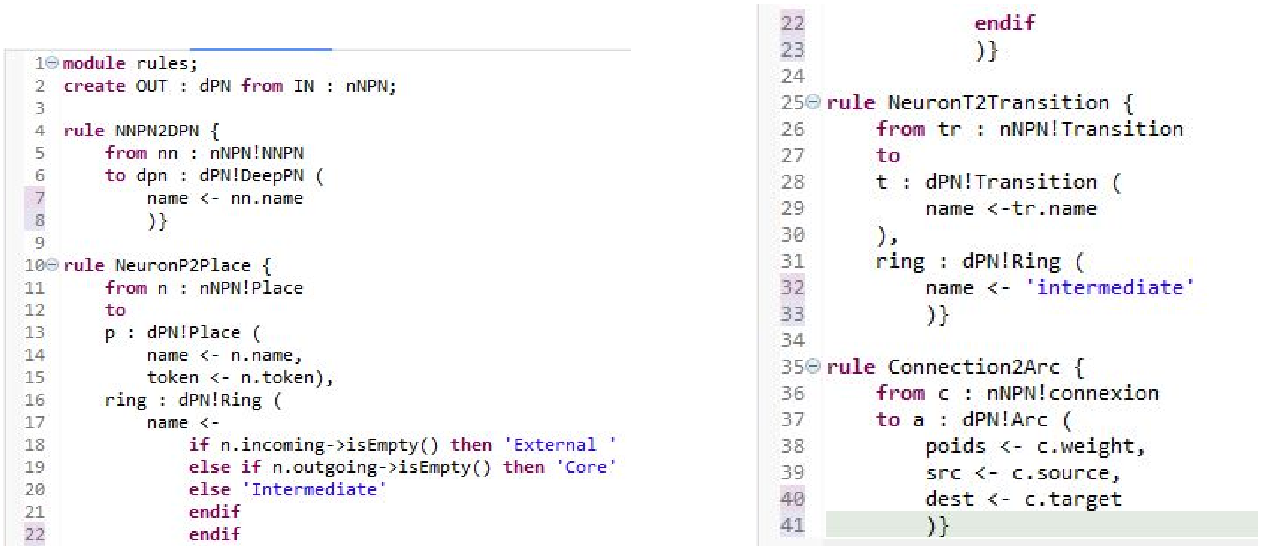
- **Class `NNPN`**:
  The root element of the model. It includes:
  - A `name` attribute of type `EString`.
  - A collection of `Layers` elements.
  - A collection of `Connexion` elements.
- **Class `Layers`**:
  Represents a layer in the neural network structure. It includes:
  - A `name` attribute of type `EString`.
  - A collection of `Neuron` elements.
- **Class `Neuron`** (abstract):
  A generic unit within a layer that can represent either a `Place` or a `Transition`. It includes:
  - A `name` attribute of type `EString`.
  - One or more incoming arcs.
  - One or more outgoing arcs.
- **Class `Place`** (inherits from `Neuron`):
  Represents a state or condition within the system. It includes:
  - A `token` attribute of type `EInt`.
- **Class `Transition`** (inherits from `Neuron`):
  Represents an event or action.
- **Class `Connexion`**:
  Represents a weighted link between two neurons (analogous to arcs in Petri nets or synapses in neural networks). It includes:
  - A `weight` attribute of type `EInt`.
  - References to `source` and `target` `Neuron`.

3.2.2 Meta Model The Deep Petri Net (DPN)

Remark. As for the Deep Petri net model, it corresponds to the one previously described under the referenced section. "The Meta Model The Deep Petri Net (DPN)".

3.2.3 Transformation

The ATL code shown in Figure 3.14 defines the transformation rules for converting the Neural Network of Petri Net (NNPN) model into a Deep Petri Net model.



```
module rules;
create OUT : dPN from IN : nNPN;

rule NNPN2DPN {
    from nn : nNPN!NNPN
    to dpn : dPN!DeepPN (
        name <- nn.name
        )}

rule NeuronP2Place {
    from n : nNPN!Place
    to
    p : dPN!Place (
        name <- n.name,
        token <- n.token),
    ring : dPN!Ring (
        name <-
            if n.incoming->isEmpty() then 'External '
            else if n.outgoing->isEmpty() then 'Core'
            else 'Intermediate'
            endif
            endif
```

(a)

```
            endif
        )}

rule NeuronT2Transition {
    from tr : nNPN!Transition
    to
    t : dPN!Transition (
        name <-tr.name
    ),
    ring : dPN!Ring (
        name <- 'intermediate'
        )}

rule Connection2Arc {
    from c : nNPN!connexion
    to a : dPN!Arc (
        poids <- c.weight,
        src <- c.source,
        dest <- c.target
        )}
```

(b)

Figure 3.14: ATL transformation rules from NNPN to Deep Petri Net

Description of the ATL Rules:

- **Rule 1: `NNPN2DPN`**
  This rule transforms the root element of the source NNPN model (Neural Network of Petri Net) into the root element of the target DPN model (Deep Petri Net).
  - The `name` attribute is directly copied from the source model.
- **Rule 2: `Neuron2Place`**
  This rule converts a neuron of type `Place` from the NNPN model into a `Place` in the DPN model.
  - The `name` and `token` attributes are transferred.
  - A `Ring` is created and assigned based on the neuron's connections:
    - ∗ No incoming connections → `External`
    - ∗ No outgoing connections → `Core`
    - ∗ Both incoming and outgoing connections → `Intermediate`
- **Rule 3: `NeuronT2Transition`**
  This rule transforms a neuron of type `Transition` into a `Transition` in the DPN model.
  - The `name` attribute is copied directly.
  - The transition is placed into a `Ring` named `"Intermediate"`.

- **Rule 4: `Connection2Arc`**
  This rule converts a `connexion` element from the NNPN model into an `Arc` in the DPN model.
  - The `weight` is copied as `poids`.
  - The `source` and `target` references are transferred to the arc.

Conclusion

In this chapter, we have explored two distinct model transformation approaches aimed at bridging classical Petri Nets with Deep Petri Nets through Model-Driven Engineering techniques. By utilizing the ATL language and EMF-based metamodels, we demonstrated how formal structures can be progressively enriched with neural-inspired capabilities. These transformations enable the transition from static, rule-based models to more dynamic and adaptive systems. Ultimately, this work highlights the potential of combining formal modeling with machine learning concepts to support the simulation, analysis, and design of complex systems in a structured and automated way.

Chapter4

Case Studies

Introduction

In this chapter, we illustrate the practical application of our model transformation approaches by conducting two case studies on intelligent systems. These case studies aim to demonstrate how our methodology enhances the modeling, simulation, and analysis of complex and dynamic behaviors. The selected systems a Smart Traffic Management System and an Automated Production Workshop represent real-world scenarios where intelligent coordination and decision-making are crucial. Through visual representations, transformation steps, and resulting models, we provide evidence of the effectiveness and adaptability of our proposed methodology.

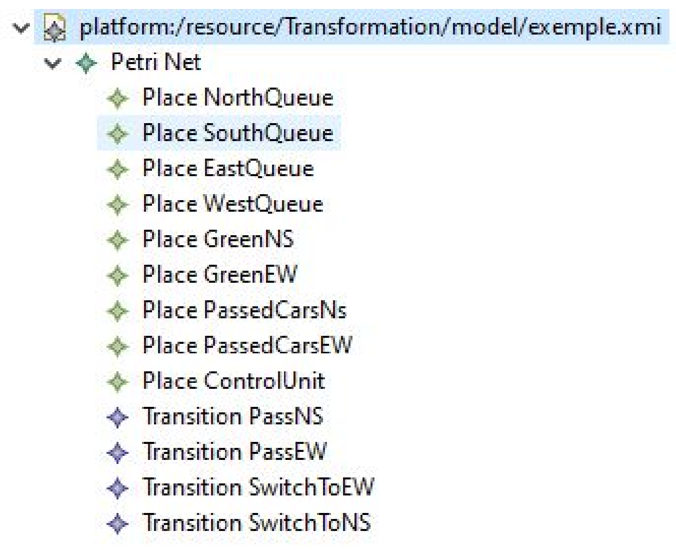
4.1 First Case Study

To demonstrate how our transformation process improves the modeling, simulation, and analysis of complex and dynamic behaviors in intelligent systems, we applied our approaches to a real-world example: **Smart Traffic Management System (STMS)** is an intelligent infrastructure designed to monitor and regulate traffic flow in urban environments.

4.1.1 First Approach

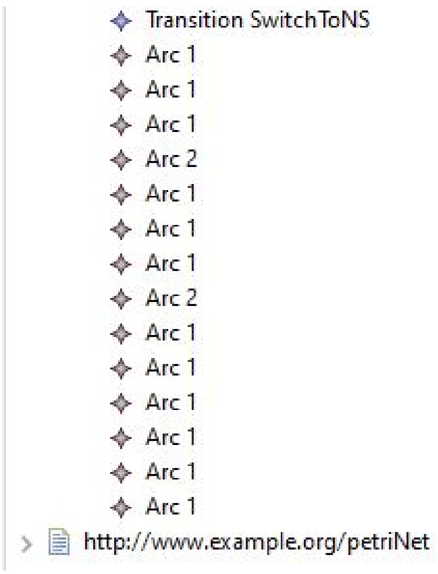
- **Transformation From Petri net into Neural Network**

  In this transformation, We generate an XMI file that encodes the Petri net representation of the Smart Traffic Management System " Figures 4.1 ".



```
platform:/resource/Transformation/model/exemple.xmi
  Petri Net
    Place NorthQueue
    Place SouthQueue
    Place EastQueue
    Place WestQueue
    Place GreenNS
    Place GreenEW
    Place PassedCarsNs
    Place PassedCarsEW
    Place ControlUnit
    Transition PassNS
    Transition PassEW
    Transition SwitchToEW
    Transition SwitchToNS
```

(a)

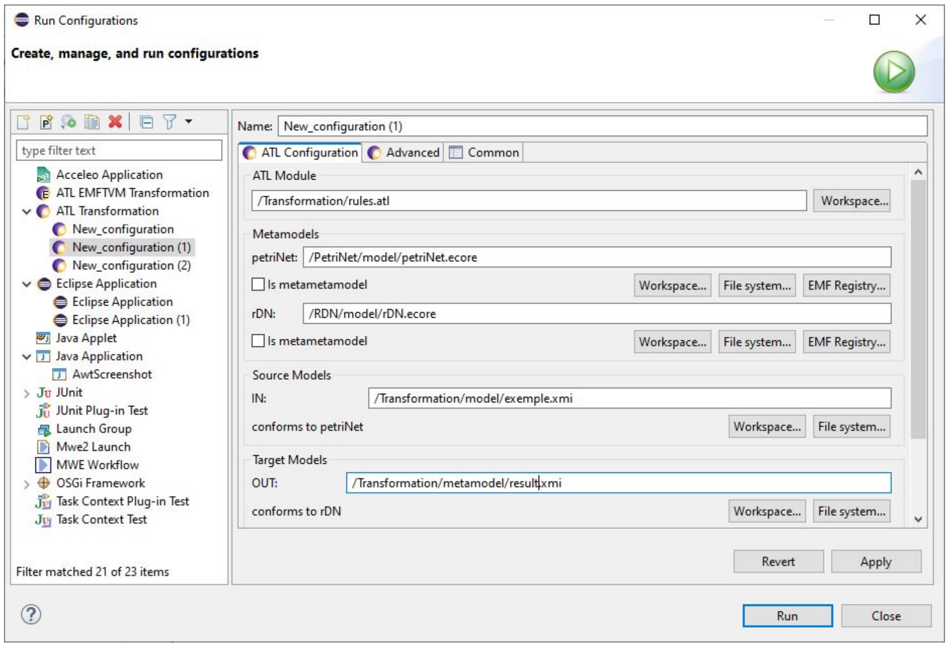


```
    Transition SwitchToNS
    Arc 1
    Arc 1
    Arc 1
    Arc 2
    Arc 1
    Arc 1
    Arc 1
    Arc 2
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
http://www.example.org/petriNet
```

(b)

Figure 4.1: Smart Traffic Management System modeled as a Petri Net

Then we apply ATL rules to transform this representation into a neural network model.The execution of the transformation process is shown in Figure 4.2.



Name: New_configuration (1)

ATL Module
/Transformation/rules.atl

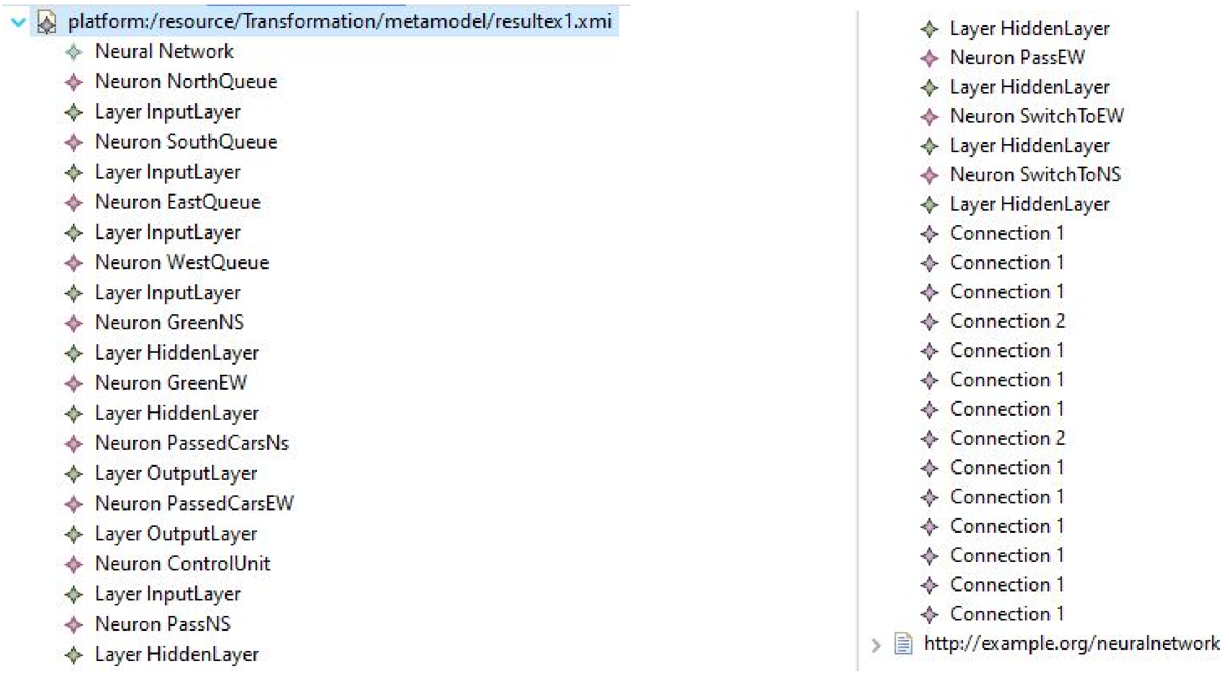
Metamodels
petriNet: /PetriNet/model/petriNet.ecore
☐ Is metametamodel
rDN: /RDN/model/rDN.ecore
☐ Is metametamodel

Source Models
IN: /Transformation/model/exemple.xmi
conforms to petriNet

Target Models
OUT: /Transformation/metamodel/result.xmi
conforms to rDN

Figure 4.2: Execution of ATL transformation from Petri Net to Neural Network

The resulting Neural Network model is saved in XMI file. Figures 4.3 show the generated model.



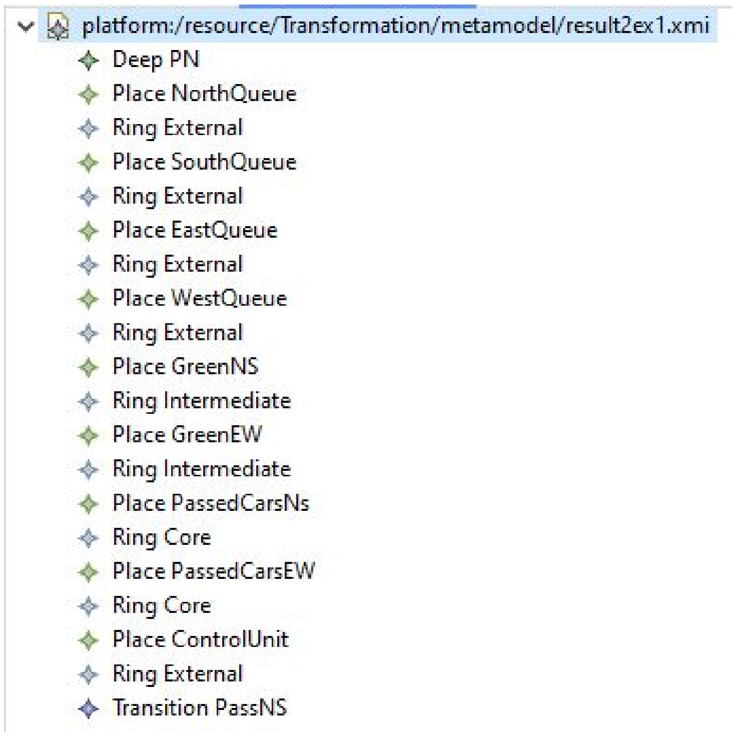
platform:/resource/Transformation/metamodel/resultex1.xmi
Neural Network
Neuron NorthQueue
Layer InputLayer
Neuron SouthQueue
Layer InputLayer
Neuron EastQueue
Layer InputLayer
Neuron WestQueue
Layer InputLayer
Neuron GreenNS
Layer HiddenLayer
Neuron GreenEW
Layer HiddenLayer
Neuron PassedCarsNs
Layer OutputLayer
Neuron PassedCarsEW
Layer OutputLayer
Neuron ControlUnit
Layer InputLayer
Neuron PassNS
Layer HiddenLayer

Layer HiddenLayer
Neuron PassEW
Layer HiddenLayer
Neuron SwitchToEW
Layer HiddenLayer
Neuron SwitchToNS
Layer HiddenLayer
Connection 1
Connection 1
Connection 1
Connection 2
Connection 1
Connection 1
Connection 1
Connection 2
Connection 1
Connection 1
Connection 1
Connection 1
Connection 1
Connection 1
http://example.org/neuralnetwork

(a) (b)

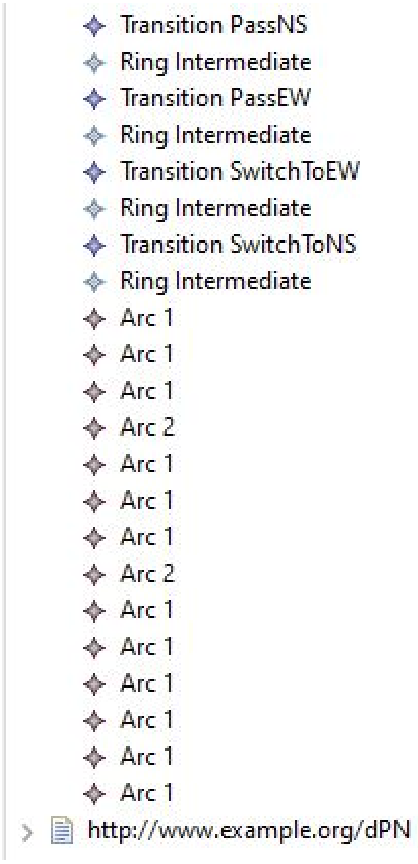
Figure 4.3: Generated Neural Network model

- **Transformation From The Resulting into Deep petri net** In this transformation, the model generated by the initial transformation is used as input, and ATL rules are applied to produce the final Deep Petri Net model



platform:/resource/Transformation/metamodel/result2ex1.xmi
Deep PN
Place NorthQueue
Ring External
Place SouthQueue
Ring External
Place EastQueue
Ring External
Place WestQueue
Ring External
Place GreenNS
Ring Intermediate
Place GreenEW
Ring Intermediate
Place PassedCarsNs
Ring Core
Place PassedCarsEW
Ring Core
Place ControlUnit
Ring External
Transition PassNS

(a)



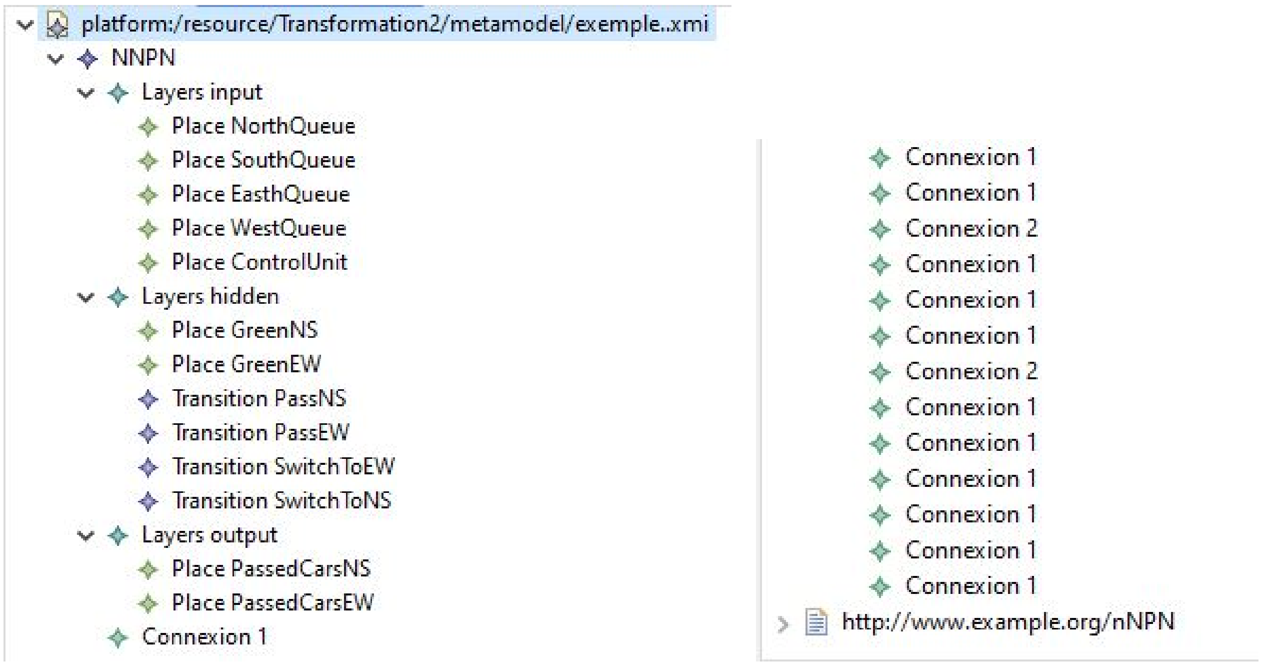
Transition PassNS
Ring Intermediate
Transition PassEW
Ring Intermediate
Transition SwitchToEW
Ring Intermediate
Transition SwitchToNS
Ring Intermediate
Arc 1
Arc 1
Arc 1
Arc 2
Arc 1
Arc 1
Arc 1
Arc 2
Arc 1
Arc 1
Arc 1
Arc 1
Arc 1
Arc 1
http://www.example.org/dPN

(b)

Figure 4.4: Generated Deep Petri Net model

4.1.2 Second Approach

In this approach, We generate an XMI file that encodes the Neural Network of Petri Net(NNPN) representation of the Smart Traffic Management System.



```
v platform:/resource/Transformation2/metamodel/exemple..xmi
  v NNPN
    v Layers input
        Place NorthQueue
        Place SouthQueue
        Place EasthQueue
        Place WestQueue
        Place ControlUnit
    v Layers hidden
        Place GreenNS
        Place GreenEW
        Transition PassNS
        Transition PassEW
        Transition SwitchToEW
        Transition SwitchToNS
    v Layers output
        Place PassedCarsNS
        Place PassedCarsEW
      Connexion 1
```

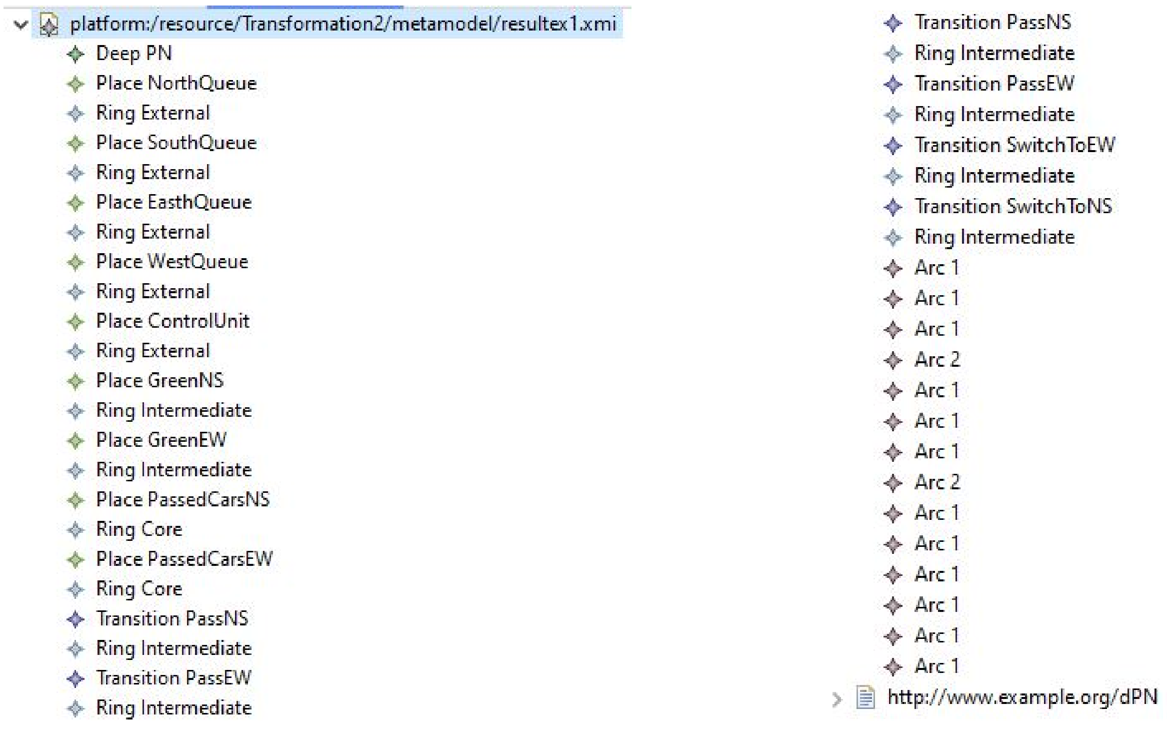
(a)

```
    Connexion 1
    Connexion 1
    Connexion 2
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 2
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
> http://www.example.org/nNPN
```

(b)

Figure 4.5: Smart Traffic Management System modeled as a NNPN

Then we apply ATL transformation rules to generate the final Deep Petri Net model.



```
v platform:/resource/Transformation2/metamodel/resultex1.xmi
    Deep PN
    Place NorthQueue
    Ring External
    Place SouthQueue
    Ring External
    Place EasthQueue
    Ring External
    Place WestQueue
    Ring External
    Place ControlUnit
    Ring External
    Place GreenNS
    Ring Intermediate
    Place GreenEW
    Ring Intermediate
    Place PassedCarsNS
    Ring Core
    Place PassedCarsEW
    Ring Core
    Transition PassNS
    Ring Intermediate
    Transition PassEW
    Ring Intermediate
```

(a)

```
    Transition PassNS
    Ring Intermediate
    Transition PassEW
    Ring Intermediate
    Transition SwitchToEW
    Ring Intermediate
    Transition SwitchToNS
    Ring Intermediate
    Arc 1
    Arc 1
    Arc 1
    Arc 2
    Arc 1
    Arc 1
    Arc 1
    Arc 2
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
> http://www.example.org/dPN
```

(b)

Figure 4.6: Generated Deep Petri Net model

4.2 Second Case Study

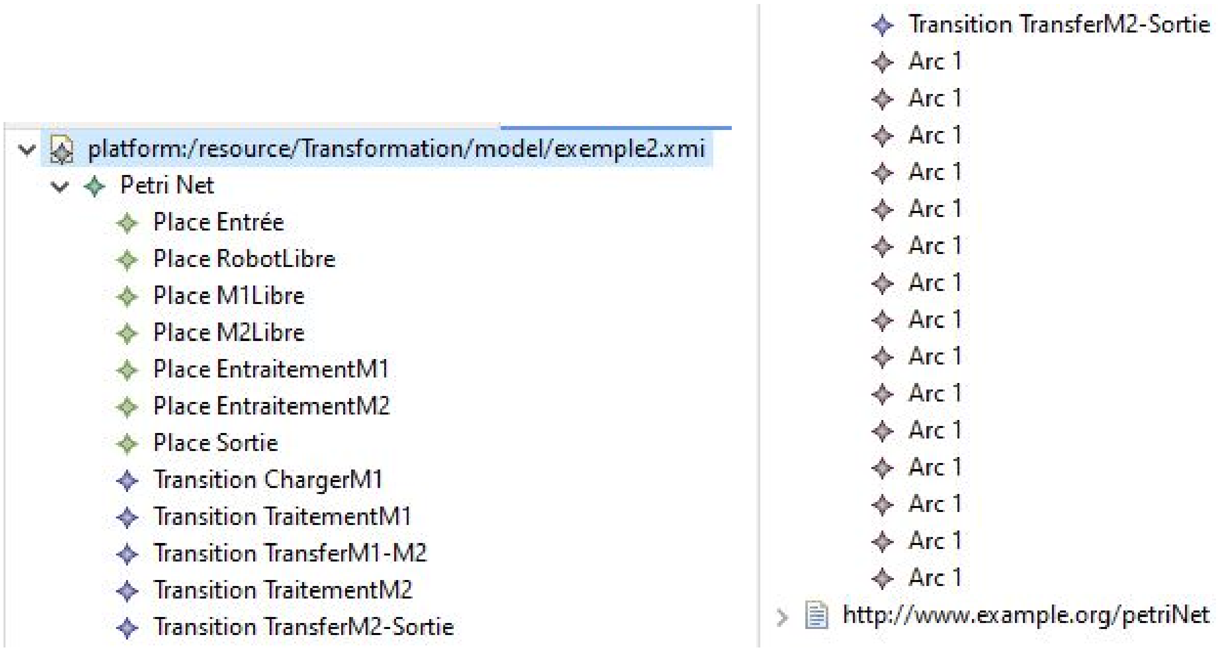
To demonstrate how our transformation process improves the modeling, simulation, and analysis of complex and dynamic behaviors in intelligent systems, we applied our approaches to: Automated Production Workshop ,is a manufacturing scenario that involves coordination between machines, robots, and a production line.

4.2.1 First Approach

Transformation From Petri Net into Neural Network

For this transformation we generate an XMI file "Figures 4.7" , that encodes the Petri net representation of the Automated Production Workshop.

Then we apply ATL rules to transform this representation into a neural network model



```
platform:/resource/Transformation/model/exemple2.xmi
  Petri Net
    Place Entrée
    Place RobotLibre
    Place M1Libre
    Place M2Libre
    Place EntraitementM1
    Place EntraitementM2
    Place Sortie
    Transition ChargerM1
    Transition TraitementM1
    Transition TransferM1-M2
    Transition TraitementM2
    Transition TransferM2-Sortie
```

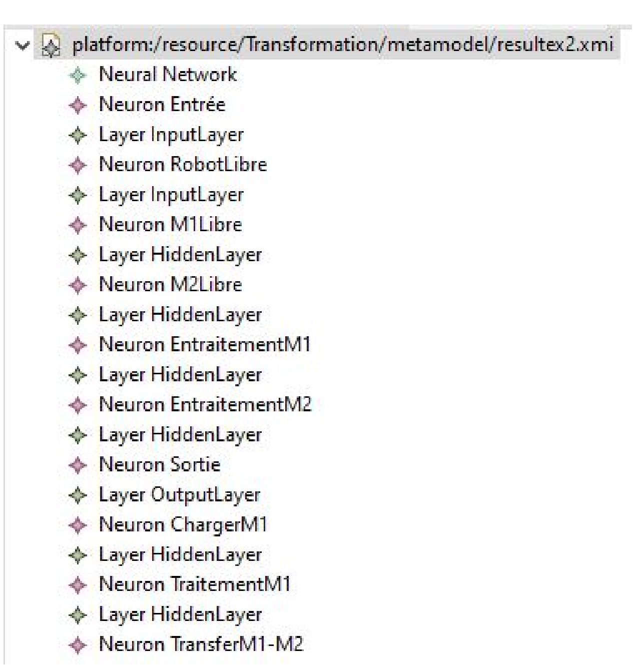
(a)

```
    Transition TransferM2-Sortie
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
    Arc 1
http://www.example.org/petriNet
```

(b)

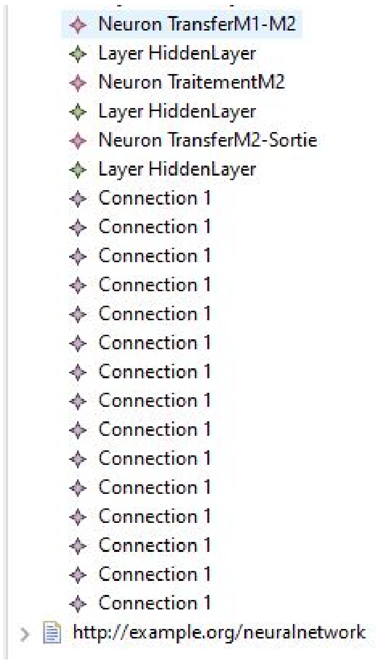
Figure 4.7: Automated Production Workshop modeled as a Petri Net

The resulting is saved in the XMI file. Figures 4.8 show the generated model.



platform:/resource/Transformation/metamodel/resultex2.xmi
- Neural Network
- Neuron Entrée
- Layer InputLayer
- Neuron RobotLibre
- Layer InputLayer
- Neuron M1Libre
- Layer HiddenLayer
- Neuron M2Libre
- Layer HiddenLayer
- Neuron EntraitementM1
- Layer HiddenLayer
- Neuron EntraitementM2
- Layer HiddenLayer
- Neuron Sortie
- Layer OutputLayer
- Neuron ChargerM1
- Layer HiddenLayer
- Neuron TraitementM1
- Layer HiddenLayer
- Neuron TransferM1-M2

(a)



- Neuron TransferM1-M2
- Layer HiddenLayer
- Neuron TraitementM2
- Layer HiddenLayer
- Neuron TransferM2-Sortie
- Layer HiddenLayer
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1
- Connection 1

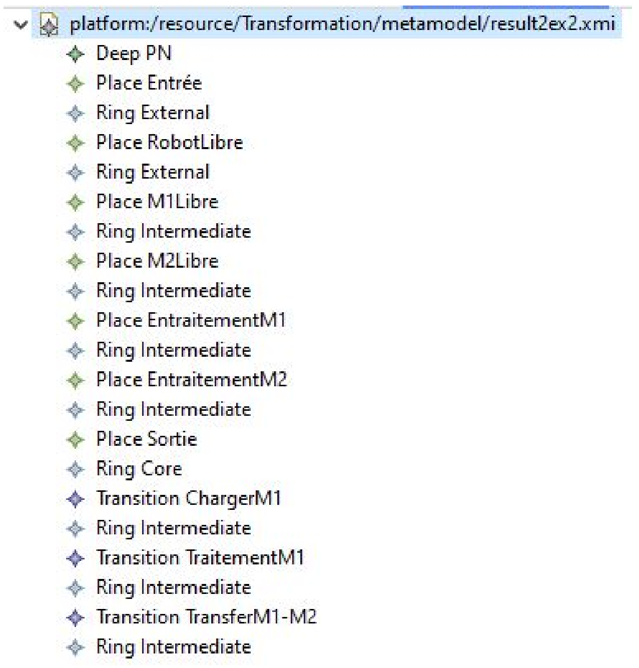
http://example.org/neuralnetwork

(b)

Figure 4.8: Generated Neural Network model

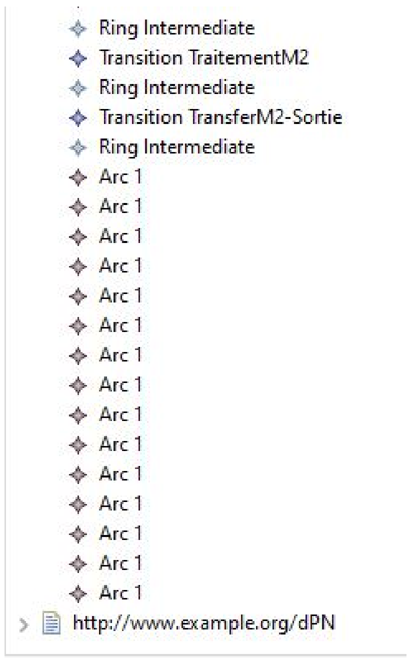
Transformation From The Resulting into Deep petri net

In this transformation, the model generated by the initial transformation is used as input, and ATL rules are applied to produce the final Deep Petri Net model.



platform:/resource/Transformation/metamodel/result2ex2.xmi
- Deep PN
- Place Entrée
- Ring External
- Place RobotLibre
- Ring External
- Place M1Libre
- Ring Intermediate
- Place M2Libre
- Ring Intermediate
- Place EntraitementM1
- Ring Intermediate
- Place EntraitementM2
- Ring Intermediate
- Place Sortie
- Ring Core
- Transition ChargerM1
- Ring Intermediate
- Transition TraitementM1
- Ring Intermediate
- Transition TransferM1-M2
- Ring Intermediate

(a)



- Ring Intermediate
- Transition TraitementM2
- Ring Intermediate
- Transition TransferM2-Sortie
- Ring Intermediate
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1
- Arc 1

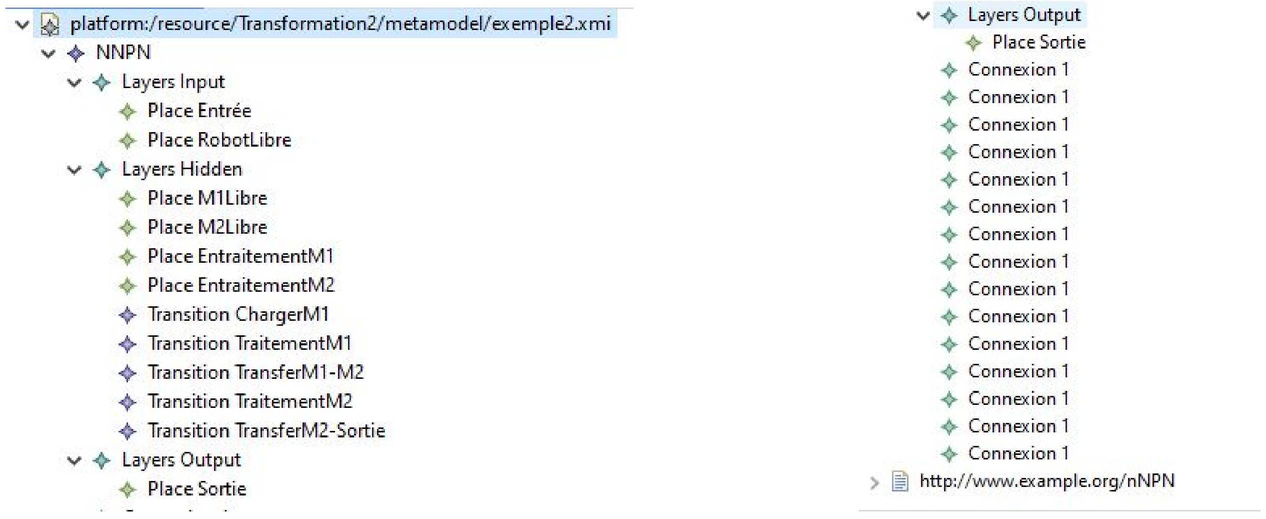
http://www.example.org/dPN

(b)

Figure 4.9: Generated Deep Petri Net model

4.2.2 Second Approach

In this approach, We generate an XMI file that encodes the Neural Network of Petri Net (NNPN) representation of the Automated Production Workshop.



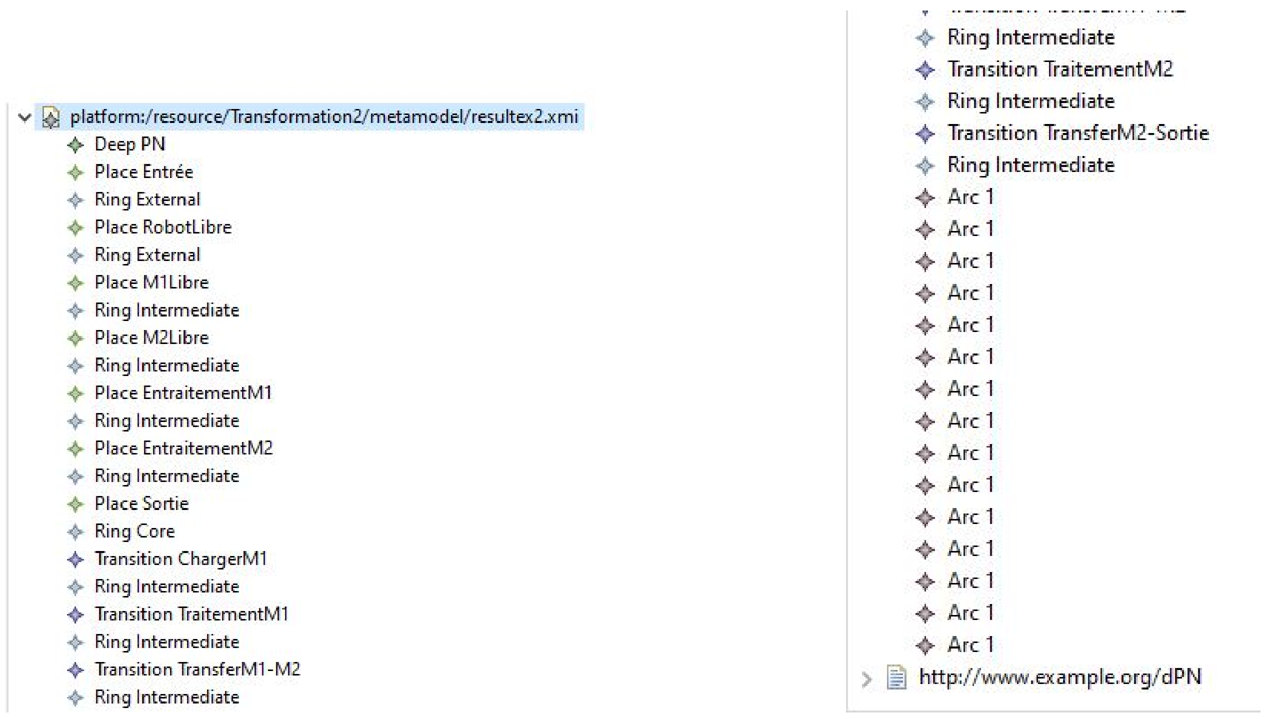
```
platform:/resource/Transformation2/metamodel/exemple2.xmi
  NNPN
    Layers Input
      Place Entrée
      Place RobotLibre
    Layers Hidden
      Place M1Libre
      Place M2Libre
      Place EntraitementM1
      Place EntraitementM2
      Transition ChargerM1
      Transition TraitementM1
      Transition TransferM1-M2
      Transition TraitementM2
      Transition TransferM2-Sortie
    Layers Output
      Place Sortie
```

```
    Layers Output
      Place Sortie
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
    Connexion 1
> http://www.example.org/nNPN
```

(a) (b)

Figure 4.10: Smart Traffic Management System modeled as a NNPN

Then we apply ATL transformation rules to generate the final Deep Petri Net model.



```
platform:/resource/Transformation2/metamodel/resultex2.xmi
  Deep PN
  Place Entrée
  Ring External
  Place RobotLibre
  Ring External
  Place M1Libre
  Ring Intermediate
  Place M2Libre
  Ring Intermediate
  Place EntraitementM1
  Ring Intermediate
  Place EntraitementM2
  Ring Intermediate
  Place Sortie
  Ring Core
  Transition ChargerM1
  Ring Intermediate
  Transition TraitementM1
  Ring Intermediate
  Transition TransferM1-M2
  Ring Intermediate
```

```
  Ring Intermediate
  Transition TraitementM2
  Ring Intermediate
  Transition TransferM2-Sortie
  Ring Intermediate
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
  Arc 1
> http://www.example.org/dPN
```

(a) (b)

Figure 4.11: Generated Deep Petri Net model

Conclusion

This chapter has demonstrated the feasibility and utility of our approaches through two representative case studies Smart Traffic Management System and an Automated Production Workshop , we validated the capacity of our methodology to bridge Petri nets and neural-based architectures, resulting in formal, analyzable Deep Petri Nets.

General Conclusion

Conclusion

In this memory, we presented a novel approaches that bridges the gap between symbolic modeling and subsymbolic learning by integrating the formal rigor of Petri Nets with the adaptive capabilities of neural networks. This integration led to the formulation of the Deep Petri Net (DPN) model a hybrid framework capable of representing both structural logic and dynamic learning processes.

Our methodology was structured around two main transformation approaches. The first approach defines three metamodels: the first for classical Petri Nets, the second for artificial neural networks, and the third for Deep Petri Nets. These metamodels were developed using the Eclipse Modeling Framework (EMF), and transformations between them were implemented using the Atlas Transformation Language (ATL). This enabled a step-by-step conversion from symbolic models to hybrid intelligent systems.

The second approach introduces a unified intermediate metamodel, the Neural Network Petri Net (NNPN), which directly integrates neural components into the Petri Net structure. This unified model is subsequently transformed into a Deep Petri Net using ATL rules. The direct embedding approach simplifies the transformation chain while maintaining consistency and cohesion.

Future Works

Looking ahead, several promising directions can be explored to improve and extend the proposed framework.

First, enhancing the scalability and performance of the DPN framework is essential. This can be achieved by optimizing transformation rules and exploring modularization strategies to partition complex models into smaller, manageable components.

Second, the framework could be extended to support additional learning paradigms such as reinforcement learning and unsupervised learning. These capabilities would increase the adaptability of the system in diverse scenarios.

Finally, improving the explainability of the decision-making processes within Deep Petri Nets will be crucial, particularly for applications in safety-critical domains such as healthcare, autonomous vehicles, or industrial automation.

In conclusion, the Deep Petri Net framework provides a strong foundation for unifying symbolic reasoning with adaptive learning. Through its dual transformation approaches it opens new possibilities for the design of intelligent, explainable, and traceable systems. We believe this approach represents a valuable contribution to the field of neuro-symbolic artificial intelligence and hybrid system modeling.

Bibliography

[1] T. Murata (1989), 'Petri Nets: Properties, Analysis and Applications," *IEEE Transactions on Communications*, vol. 77, no. 4, pp. 541-580.

[2] J. Desel and W. Reisig (2001), *Lectures on Petri Nets I: Basic Models*.

[3] W. M. P. van der Aalst (2020), *Process Mining: Data Science in Action* (2nd ed.).

[4] X. Xu, Y. Lu, B. Vogel-Heuser, and L. Wang (2021), 'Industrial AI and Digital Transformation for Smart Manufacturing: A Review," *Engineering*, vol. 7, no. 6, pp. 738-752.

[5] D. Fahland and M. Weidlich (2023), 'Conformance Checking and Process Enhancement Using Petri Nets," *Foundations of Computing and Decision Sciences*, vol. 48, no. 1, pp. 1-27.

[6] Y. LeCun, Y. Bengio, and G. Hinton (2015), 'Deep learning," *Nature*, vol. 521, pp. 436-444.

[7] I. Goodfellow, Y. Bengio, and A. Courville (2016), *Deep Learning*, MIT Press.

[8] Y. Liu, H. Zhang, and L. Wang (2020), 'Deep Fuzzy Petri Nets: A Transparent Neural Reasoning Framework," *Knowledge-Based Systems*, vol. 192, p. 105-354.

[9] W. Reisig (2013), *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*.

[10] R. David and H. Alla (2010), *Discrete, Continuous, and Hybrid Petri Nets*.

[11] W. M. P. van der Aalst (2016), *Process Mining: Data Science in Action* (2nd ed.).

[12] M. Aouag (2023), *Chapter 02: Petri Nets (PN)*. Available at: `https://elearning.centre-univ-mila.dz/a2024/pluginfile.php/69978/mod_resource/content/0/Chapitre%2002_%20Les%20R%C3%A9seaux%20de%20P%C3%A9tri%20_RDP.pdf` [Consulted on May 10, 2025]

[13] J. Desel and J. Esparza (1995), *Free Choice Petri Nets*, Cambridge University Press.

[14] R. Zhu, Q. Ban, and X. Cong (2019), 'Modules of Petri Nets and New Petri Net Structure: Arcs with a Weighted Function Set," *Journal of Intelligent & Fuzzy Systems*, vol. 36, no. 5, pp. 4567-4578.

[15] E. Popova (2024), 'Controlling Petri Net Behavior Using Priorities for Transitions,".

[16] P. Janar, J. Leroux, and J. Valek (2025), *Structural Liveness of Conservative Petri Nets*, arXiv preprint arXiv:2503.11590.

[17] Y. Zhang and M. Zhou (2021), 'Time-Based Deadlock Prevention for Petri Nets," *Automatica*, vol. 130, p. 109-716.

[18] K. Barkaoui and G. Balbo (2021), *Timed Petri Nets: Modeling, Performance Evaluation, and Applications*.

[19] Q. He and H. Gharavi (2022), *Stochastic Petri Nets and Their Applications in Computer Systems and Networks*, Wiley-IEEE Press.

[20] M. A. Ribeiro and A. R. Silva (2021), *Colored Petri Nets in Business Process Modeling and Performance Evaluation*.

[21] M. M. Hammad (2024), *Artificial Neural Network and Deep Learning: Fundamentals and Theory*.

[22] A. Boulmerka (2022), *Chapitre 07 : Les réseaux de neurones (RN)*. Consulté à : `https://elearning.centre-univ-mila.dz/a2024/pluginfile.php/3300/mod_resource/content/6/chap06.reseaux_neurones.pdf`[Consulted on May 11, 2025]

[23] I. Goodfellow, Y. Bengio, and A. Courville (2016), *Deep Learning*, MIT Press. [Chapter 6]

[24] F. Chollet (2021), *Deep Learning with Python* (2nd ed.), Manning Publications. [Chapter 5]

[25] I. Goodfellow, Y. Bengio, and A. Courville (2016), *Deep Learning*, MIT Press. [Chapter 10]

[26] I. Goodfellow, Y. Bengio, and A. Courville (2016), *Deep Learning*, MIT Press. [Chapter 20]

[27] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu (2021), 'A Comprehensive Survey on Graph Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*.

[28] Y.-N. Lin, T.-Y. Hsieh, C.-Y. Yang, V. R. L. Shen, T. T.-Y. Juang, and W.-H. Chen (2020), 'Deep Petri Nets of Unsupervised and Supervised Learning," *Measurement and Control*, vol. 53, no. 78, pp. 1061-1072.

[29] H. Qi, M. Guang, J. Wang, and C. Jiang (2023), 'A Perspective on Petri Net Learning."

[30] W. He, W. Chen, Y. Wang, and P. Liu (2021), 'Deep Petri Nets: A Neural-symbolic Integration for Modeling and Learning Complex Systems," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 12, pp. 5564-5578.

[31] Y. Zhang, Y. Zhang, and X. Liu (2020), 'Deep Petri Nets: A Novel Framework for Modeling and Learning Dynamic Systems," *Neurocomputing*, vol. 417, pp. 347-359.

[32] Object Management Group (2003), *MDA Guide Version 1.0.1*, OMG Document omg/2003-06-01.

[33] I. Sommerville (2016), *Software Engineering* (10th ed.).

[34] M. Brambilla, J. Cabot, and M. Wimmer (2012), *Model-Driven Software Engineering in Practice*, Morgan & Claypool Publishers.

[35] C. Atkinson and T. Kühne (2003), ‘Modeling concepts for information systems engineering,” In *Conceptual Modeling*, pp. 513-524.

[36] T. Kühne (2006), ‘Matters of (meta-) modeling,” *Software & Systems Modeling*, vol. 5, no. 4, pp. 369-385.

[37] Université Badji Mokhtar Annaba (2021), *Introduction à lIngénierie Dirigée par les Modèles*, Chapitre 1. `https://elearning-facsci.univ-annaba.dz/pluginfile.php/44809/mod_resource/content/1/chapitre%201.pdf` [Consulted on May 16, 2025]

[38] A. Hettab (2009), *De M-UML vers les réseaux de Pétri "Nested Nets"* : une approche basée sur la transformation de graphes, Thèse de doctorat, Université de Mentouri Constantine, pp. 58-74.

[39] T. Mens and P. Van Gorp (2006), ‘A taxonomy of model transformation,” *Software & Systems Modeling*, vol. 5, no. 1, pp. 35-51.

[40] K. Czarnecki and S. Helsen (2006), ‘Classification of model transformation approaches,” In *OOPSLA Workshop on Generative Techniques in the Context of MDA*.

[41] M. Amroune (2014), *Vers une approche orientée aspect dingénierie des besoins dans les organisations multi-entreprises*, Thèse de doctorat, Université de Toulouse, p. 50.

[42] M. Wimmer, G. Kappel, et al. (2017), *Model-Driven Software Engineering in Practice* (2nd ed.), Morgan & Claypool Publishers.

[43] M. Bendiaf (2018), *Spécification et vérification des systèmes embarqués temps réel en utilisant la logique de réécriture*, Thèse de doctorat, Université Mohamed Khider Biskra, pp. 78-79.

[44] S. Meghzili (2021), ‘Transformation de Graphes Chapitre 03,” *Cours Ingénierie des Logiciels*, Ministère de lEnseignement Supérieur et de la Recherche Scientifique, Centre Universitaire de Mila, Laboratoire MISC, Université Constantine 2 Abdelhamid Mehri.

[45] L. Fritsche, J. Kosiol, A. Schürr, and G. Taentzer (2020), ‘Avoiding Unnecessary Information Loss: Correct and Efficient Model Synchronization Based on Triple Graph Grammars,” *arXiv preprint arXiv:2005.14510*.

[46] H. Giese and R. Wagner (2009), ‘From model transformation to incremental bidirectional model synchronization,” *Software & Systems Modeling*, vol. 8, no. 1, pp. 21-43.

[47] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro (2008), *EMF: Eclipse Modeling Framework* (2nd ed.), Addison-Wesley.

[48] I. Ben Hmida and I. Bouassida Rodriguez (2021), ‘Comparative Study of Model Transformation Approaches: ATL and TGG,” *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 1, pp. 520-528.

[49] F. Jouault and I. Kurtev (2006), 'Transforming Models with ATL," In *Satellite Events at the MoDELS 2005 Conference*, pp. 128-138.

[50] Y. Liu et al. (2021), 'Deep Fuzzy Petri Nets: A Hybrid Framework for Explainable AI," *Neural Computing and Applications*, vol. 33, pp. 10521-10535.

[51] Q. Zhao et al. (2020), 'An Improved Fuzzy Petri Net for Uncertain Knowledge Representation," *Applied Intelligence*, vol. 50, pp. 4123-4135.

[52] H. Wang and Y. Zhao (2019), 'A Fuzzy Petri Net-Based Intelligent Framework for Real-Time Decision Support," *Expert Systems with Applications*, vol. 127, pp. 143-154.

[53] F. Kordon, H. Garavel, and J. Thierry-Mieg (2021), 'Formalization of AI Systems with Petri Nets and MDE," *Journal of Systems Architecture*, vol. 121, p. 102-688.

[54] L. Zhang et al. (2022), 'Deep Reinforcement Petri Nets for Autonomous Decision-Making," *IEEE Access*, vol. 10, pp. 21890-21904.

[55] H. Yin et al. (2023), 'Petri Net-Based Interpretability for Deep Learning Models," *Information Sciences*, vol. 636, pp. 230-245.

[56] Y. Zhao, H. Liu, M. Chen, and X. Wang (2024), 'Petri-net-based deep reinforcement learning for real-time scheduling of automated manufacturing systems," *Journal of Manufacturing Systems*, vol. 74, pp. 995-1008.

[57] C. A. Petri (1962), *Communication with Automata*, Technical Report No. RADC-TR-65-377, Institut für Instrumentelle Mathematik, Bonn.

[58] K. Jensen (1997), *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*.

[59] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis (1995), *Modeling with Generalized Stochastic Petri Nets*, Wiley.

[60] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner (1998), "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324.

[61] S. Hochreiter and J. Schmidhuber (1997), "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780.

[62] Z. C. Lipton (2016), "The Mythos of Model Interpretability," *arXiv preprint arXiv:1606.03490*.

[63] W. Zhang (2000), "Fuzzy Petri Nets and Their Applications in Intelligent Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 30, no. 1, pp. 31–43.

[64] Y. Liu, H. Zhang, L. Wang, and J. Chen (2020), "Deep Fuzzy Petri Nets: A Hybrid Framework for Explainable Decision Making," *IEEE Access*, vol. 8, pp. 137855–137866.

[65] T. H. Tuan, C. Y. Yang, V. R. L. Shen, and W.-H. Chen (2020), "Neural Petri Nets for Modeling and Simulation of Adaptive Systems," *Journal of Intelligent & Fuzzy Systems*, vol. 38, no. 5, pp. 5631–5642.