Formalization of Deep Learning Techniques with the Why3 Proof Platform

Márcio Alexandre Mota Sousa

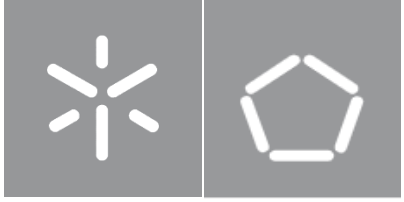
UMinho | 2021

Universidade do Minho
Escola de Engenharia

Márcio Alexandre Mota Sousa

Formalization of Deep Learning Techniques with the Why3 Proof Platform

dezembro de 2021



Universidade do Minho
Escola de Engenharia

Márcio Alexandre Mota Sousa

Formalization of Deep Learning Techniques with the Why3 Proof Platform

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Trabalho efetuado sob a orientação de
Jorge Miguel de Matos Sousa Pinto
Paulo Jorge de Sousa Azevedo

dezembro de 2021

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros

Acknowledgments

I would like to express my sincere gratitude to my supervisor and to my co-supervisor, Professor Jorge Sousa Pinto and Professor Paulo Azevedo, for all the provided support and knowledge that was invaluable not only in the elaboration of this present dissertation, but also during the whole research process associated with it.

To my parents and my brother Ruben for all the support provided during my 23 years of existence, for always believing in me and pushing me to always go a step further. This unswerving support in everything I do has been and will always be the greatest helping force that I could ask for.

To Sara for the constant support, motivation, comprehension and all the help provided during this journey, thank you for your immeasurable patience and love. And thank you for all the pointed corrections necessary on this dissertation, without which this would be a much poorer document.

And last but not least, to all the friends that contributed to who I am today, and with whom I shared many experiences throughout all these years. With a special thanks to my house colleague, Diogo, for always providing the words I was missing to make this dissertation the best it could be.

Márcio Sousa

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Formalização de Técnicas de Deep Learning com a plataforma de prova Why3

Machine Learning como um campo, parte integrante da área de Inteligência Artificial, tem crescido exponencialmente, principalmente nesta última década, onde passou de quase desconhecido pelo público em geral para a existência de carros autónomos e até robôs humanóides como o robô Sophia da Arábia Saudita. A maioria de nós agora lida com Inteligência Artificial todos os dias, em anúncios direcionados por exemplo, o que é agora a norma.

Deep Learning, um ramo específico de *Machine Learning* de onde originaram as Redes Neuronais, é vastamente utilizado no desenvolvimento de sistemas autónomos de alta complexidade. Alguns destes sistemas em particular podem ser classificados como sistemas críticos, o que traz a necessidade de fornecer alguma forma de garantia de que estes sistemas vão sempre funcionar como é suposto, uma vez que qualquer falha em sistemas desta categoria pode ter consequências graves. Isto naturalmente levanta preocupações relativas à segurança, levando a comunidade a procurar uma forma de obter tais garantias, eventualmente levando-os aos métodos de Verificação Formal para atingir os níveis de confiabilidade necessários para a adoção pública de tais sistemas.

Tem havido um interesse crescente quanto a este assunto, uma vez que as aplicações de Redes Neuronais estão em constante expansão, e muitas ferramentas de software já resultaram deste trabalho, sendo algumas dessas ferramentas analisadas nesta dissertação. Este estudo vem contribuir para esse esforço, e tem como objetivo principal a avaliação do Why3, a fim de compreender se esta ferramenta possui as características necessárias que lhe permitam juntar-se a estas ferramentas já existentes como um novo meio de verificação da correção de Redes Neuronais. Para atingir este objetivo, primeiramente criamos um *proof-of-concept* a fim de analisar se o Why3 fornece o suporte necessário para esta tarefa. Em seguida, damos um passo em frente e formalizamos uma Rede Neuronal à escala de uma aplicação real no Why3, de onde tiraremos as nossas conclusões.

Durante o trabalho sobre a formalização de Redes Neuronais, pretendemos também compilar um guia abrangente sobre Why3, desde as funcionalidades que oferece, até exemplos de como pode ser aplicado explicados passo a passo, com o objetivo de oferecer uma base de conhecimento compreensiva para qualquer pessoa interessada em explorar o Why3, contribuindo ao mesmo tempo para a escassa documentação existente sobre o Why3.

Palavras-chave: *Deep Learning*, *Machine Learning*, Rede Neuronal, Verificação Formal, Why3.

Abstract

Formalization of Deep Learning Techniques with the Why3 proof platform

Machine Learning as a field, from the realms of Artificial Intelligence, has been growing exponentially, especially in this last decade, where it went from the general public barely even hearing about it to the existence of self-driving cars and even humanoid robots like the Saudi Arabian Sophia. Most of us now deal with AI everyday, in targeted ads for example, and it has become the norm.

Deep Learning, a particular branch of Machine Learning from where Neural Networks stem, is widely used in the development of high-complexity autonomous systems. Some of these systems in particular can be classified as critical systems, which brings the necessity of providing some form of guarantee that these will always work as intended, since any failure from this category of systems can have serious consequences. This naturally raises security concerns, hence leading the community to search for a way of attaining such guarantees, eventually leading them to Formal Verification methods to achieve the necessary reliability levels for the public adoption of said systems.

There has been a growing interest regarding this matter, since the applications of Neural Networks are continuously expanding, and many software tools have already resulted from this work, some of which will be analysed in this dissertation. This study comes to contribute to this effort, and has as its main objective the evaluation of Why3, in order to understand if this tool possesses the necessary characteristics that may allow it to join these already existing tools as a new mean of verifying the correctness of Neural Networks. To achieve this objective, firstly we create a proof-of-concept in order to analyse if Why3 provides the necessary support for this task. Then we go a step further and formalize a real life application scale Neural Network in Why3, from where we will draw our conclusions.

While working on the formalization of Neural Networks, we also aim to compile a comprehensive guide on Why3, where we go from the functionalities that it provides, to examples of how it can be applied explained step by step, with the goal of offering an understandable knowledge base for anyone that may be interested in exploring Why3, while also contributing to the scarce existing documentation of Why3.

Keywords: Deep Learning, Formal Verification, Machine Learning, Neural Network, Why3.

Contents

Direitos de Autor e Condições de Utilização do Trabalho por Terceiros i

Acknowledgments ii

Statement of Integrity iii

Resumo iv

Abstract v

List of Acronyms viii

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Motivation . . . 1

1.2 Goals and Contributions . . . 2

1.3 Methodology . . . 2

1.4 Document Structure . . . 3

2 State of the Art 4

2.1 Theoretical Background . . . 4

2.1.1 Neural Networks . . . 4

2.1.2 Formal Verification . . . 5

2.2 Related Work . . . 6

2.2.1 Reluplex . . . 6

2.2.2 Marabou . . . 7

2.2.3 Verily . . . 8

3 The Why3 Proof Platform **10**
3.1 Functionalities . . . 10
3.2 Why3 Utilization Examples . . . 12
3.2.1 The Towers of Hanoi . . . 12
3.2.1.1 Implementation . . . 13
3.2.2 Mathematical Puzzle from a Dijkstra Paper . . . 17
3.2.2.1 Implementation . . . 17
3.2.3 The Area of a Triangle . . . 20

4 Preliminary Work and Results **22**
4.1 Preliminary Work . . . 22
4.1.1 Neural Network Model . . . 22
4.1.2 Formalization of the Neural Network . . . 23
4.2 Verification of the Network and its Properties . . . 24
4.3 Discussion . . . 25

5 Results and Findings **27**
5.1 Formalization of the Neural Network . . . 27
5.2 Auxiliary Generation Script . . . 28
5.3 Verification of the Network and its Properties . . . 29
5.4 Discussion . . . 30

6 Conclusions and Future Prospects **32**

Bibliography **34**

List of Acronyms

AI Artificial Inteligence. v, 1

NN Neural Network. ix, 1–11, 22, 23, 25, 27–30, 32, 33

SMT Satisfiability Modulo Theories. 6, 7, 11, 25

List of Figures

1 Example of the topology of a NN 5
2 Process of calculating the value of a neuron 5
3 ReLU activation function. 7
4 Why3ide, the Why3 graphical user-interface. 11
5 Towers of Hanoi puzzle. 13
6 Model of the Neural Network used for the proof-of-concept. 23

List of Tables

1 Logical-OR operation values. 23
2 Execution times and number of steps taken by each solver when validating both properties. 25

Chapter 1

Introduction

In this first chapter of the present dissertation, the motivation behind this work is presented, as well as the goals set to be achieved during the research. The contributions that this study resulted in are also brought forward and, finally, the methodology followed during the research, along with the structure of this document, are delineated.

1.1 Motivation

As a scientific endeavor, Machine Learning first came into being out of a quest for Artificial Intelligence and has since then risen in popularity, establishing itself as a field with ever expanding applications on this age of Big Data.

From the myriad of different types of models that this rich field has gifted the world with during its lifetime, one appears to stand out from the rest. Neural Networks (henceforth abbreviated as NNs), which arose from a particular branch of Machine Learning called Deep Learning, are computational models inspired by the biological brain, and with their artificial neurons they mimic the communication between real neurons. Thanks to their capability to learn how to perform certain tasks from finite sets of training examples and then extrapolating this knowledge to deal with previously unseen scenarios, NNs [1] have seen a considerable amount of successful applications to the most diverse areas, from cancer diagnosis [2] to speech recognition [3] and computer vision [4], just to name a few. See, for example, the case of Tesla and their self-driving cars, where a mundane task such as driving is automated resorting to Machine Learning. Even though this example is probably one of the first to come to mind when thinking of what can be achieved with the use of AI, it is also well-known that these self-driving cars have been involved in various accidents resulting of software bugs, where the software fails to recognize potential danger in its surroundings.

Self-driving cars are part of a category of software denominated Critical Systems, where the use of NNs is still limited, due to their *black box* nature. Since an error on a Critical System can imply the loss of great amounts of money or even human lives, and NNs do not provide any formal guarantee that they will behave, at all times, according to what they were trained to do, the community turned to ways of proving

NN's correctness. Thus arose the idea of applying Formal Verification methods to attain the reliability levels necessary for the public adoption of said systems.

Although neither the fields of Machine Learning nor Formal Verification are recent, the application of these software verification methods towards proving Machine Learning models is still young, especially when dealing with NNs, the case in study of this dissertation. Nevertheless, this fact has not impaired the conjugated effort of the many researchers across the world focusing on this topic, and there has been a substantial amount of work put into the verification of NNs, with some already presenting a varied selection of software tools developed specifically to this end, some of which will be discussed in the next chapter.

1.2 Goals and Contributions

Although there are already some tools developed for the purpose of formally verifying the correctness of NNs, these are still somewhat limited, and the search for new and better tools for the task at hand is still ongoing.

With this objective in mind, the research carried out in this dissertation aims to not only evaluate the Why3 proof platform's potential to be used as a NN formal verification tool, but also to contribute to the existing material about Why3 which is limited, by providing a comprehensive explanation on what this tool is capable of doing while explaining how to do it with examples. Due to some of the characteristics that Why3 features, discussed in the next chapter, everything appears to be in the right place for Why3 to join the NN verification tool group. To the best of our knowledge, even though there are tools designed for this purpose, there have been no attempts to test Why3 on the matter, so this dissertation may come to contribute by bringing Why3 into the light as a new mean of verifying the correctness of NNs.

1.3 Methodology

The path followed by the work developed in this dissertation starts by studying Why3. Being this tool the spotlight of our research, we must first get to know it, in order to figure out what could make it a potential good candidate to deal with NNs as it has been dealing with software since its creation, and also to gather as much knowledge as possible about its functioning, to be capable of explaining it in this dissertation. Then one must dive into the current state of the art regarding the verification of NNs, gathering information on how it has been done by the people already working in the area and investigating the existing tools that were created to this end.

With the study of the area done, the focus then turns to developing a formalization of a NN using WhyML, the language that Why3 provides. To achieve this, we start with a small and very simple proof-of-concept model, in order to analyze if it can be done in a small scale and then we expand the *modus operandi* used into formalizing a, greater scale, state-of-art NN that is actually being utilized in a real world application of this type of software.

Based on the results obtained from these experiments, the potential, or possibly existing issues that

may lead to the absence of it, are then analyzed with the final goal of forming conclusions about Why3's capabilities to become one of the new go-to tools when talking of verifying the correctness of NNs, pointing out all the key aspects of this tool, being them positive or negative, that lead to the final conclusion.

1.4 Document Structure

With the goal of providing a more accessible view of the organization of the present document to the reader, the structure followed by this dissertation is presented here. The chapters of this work are organized as follows:

- **State of the Art:** in this chapter the necessary background for the understanding of the dissertation is presented. Beginning with an explanation on NNs and how they operate, and following with a rundown on Formal Verification, we then proceed to provide an overview of some of the currently available tools developed for the verification of NNs.
- **The Why3 Proof Platform:** in this chapter we dive into the existing knowledge about Why3, in order to provide an easy to understand guide that may serve as a base to anyone looking to getting started with this tool, contributing to the limited existing literary material about Why3. Starting with a presentation of Why3's functionalities, we then proceed to demonstrate and explain how Why3 can be used while providing examples and explaining these step by step.
- **Preliminary Work and Results:** this chapter displays the initial work developed with the proof-of-concept verification of a NN using Why3 and the results obtained from this preliminary study, while also explaining what it entails for the work ahead.
- **Results and Findings:** this chapter goes through the process followed with the objective of formalizing a real-application-scale NN and the results obtained from such task, analyzing those same results and what findings were made based on them.
- **Conclusions and Future Prospects:** the last chapter of this document goes over the work done during the entirety of this research and what conclusions may be drawn from it, while also discussing the problems and obstacles found along the way and what prospects we have of future work to improve the final results of this dissertation.

Chapter 2

State of the Art

This chapter addresses the State of the Art of the most relevant topics concerning the work developed in this dissertation. It first offers some theoretical background that is necessary in order to understand the work developed in this study. Specifically, the provided background will target the two most important topics for the understanding of this work, NNs and Formal Verification, explaining what each of them is and how they work. Afterwards, we present an overview of some of the State of the Art tools that emerged from the specific area where those two topics converge, Formal Verification of NNs.

2.1 Theoretical Background

2.1.1 Neural Networks

Artificial NNs are computing systems conceptually based on the biological brain and its components. NNs are composed of a number of nodes, called *neurons*, which vaguely abstract the neurons of a real brain. Each of these neurons is connected to other neurons by the *edges*, and through these connections, that resemble the synapses of a brain, they are able to transmit their value to other neurons. The neurons and edges in a NN are, of course, organized, as can be seen in Figure 1. The neurons are organized in layers, which can be divided into three categories: the *input layer*, the *output layer* and the *hidden layers* which, as the name suggests, are those in between the input and the output layers. In the presented example, the NN has a total of 46 neurons, organized in 6 layers, being the first and the last, the input layer and the output layer, respectively. The blue layers are the hidden layers. To each node is assigned a *bias* value, and to each edge is assigned a *weight*. Both these values are determined when the NN is trained.

The value of a neuron n is obtained by performing a weighted sum of all the inputs received from neurons of the previous layer according to the weight of each edge, plus the bias value of n. Afterwards, the result of this sum is passed as input to an activation function, whose output will constitute the value of n. All this process is represented in Figure 2, where the values come from the nodes $x_1, x_2, ..., x_n$ of the previous layer, through the edges with weights $w_1, w_2, ..., w_n$. The weighted sum is then calculated

and added to the bias value b. Lastly, the result is passed to the activation function f, thus obtaining the value of the node.

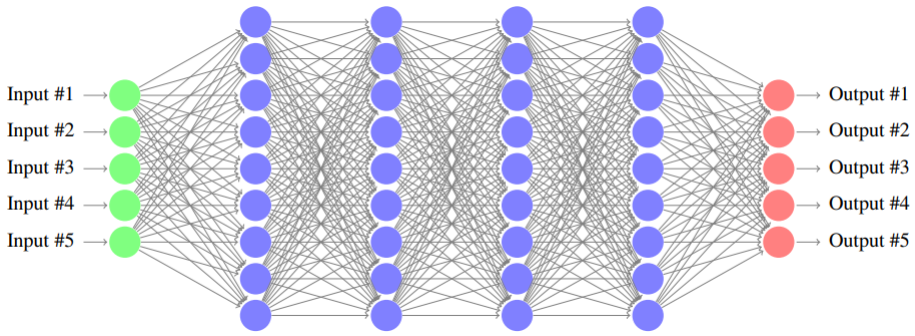


Figure 1: Example of the topology of a neural network [5].

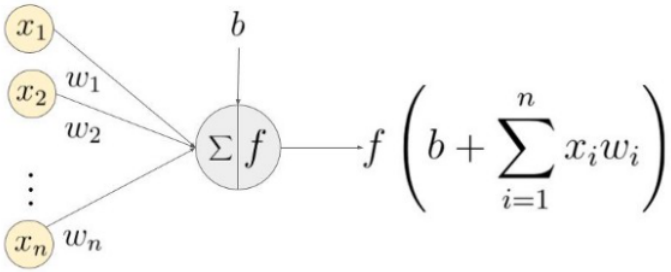


Figure 2: Process of calculating the value of a neuron [1]

Activation functions have the role of standardizing the value sent by each node to the nodes of the next layer. These functions can be, for example, the *Sigmoid* function, that always computes a number between 0 and 1, independently of the input received, or the *Rectifier Linear Unit* activation function (ReLU), that simply outputs the input without any alteration if it is a positive number, or 0 otherwise.

A very important part of NNs, and often the target of questions raised by those less familiarized with the subject, is the method which allows NNs to learn. While there are different methods applied, most of the time this capability is assured by the *Backpropagation* algorithm, popularized by David Rumelhart, Geoffrey Hinton and Ronald Williams in 1986 [6]. Simply put, after each forward iteration through the network, backpropagation executes, as the name suggests, a backwards pass, adjusting the weights and biases throughout the NN in order to obtain an output as close as possible to the desired one, in the output layer.

2.1.2 Formal Verification

In the vast field that is software development, it is not rare to see mentions to unit tests, integration tests, among many other types of tests, and these are usually used with the purpose of preventing bugs and errors in software. It is undeniable that these tests manage to find and stop many software bugs and

[1] https://learnopencv.com/wp-content/uploads/2017/10/neuron-diagram.jpg

errors before they reach production, however it would be incorrect to assume that these are able to prove their absolute absence, and this is where formal verification comes in.

The formal verification of a piece of software consists of applying mathematics and logic with the final goal of guaranteeing the correctness of the software. While the majority of software companies do not yet resort to these methods and rely simply on the usual tests, formal verification is already part of the software development process in some of the biggest companies in the area, such as Google, Meta, Amazon and Apple, along with many other companies that deal with critical software, such as avionics software for example.

Why3, the tool that we use during the entirety of this dissertation, belongs to a specific category of formal verification called Deductive Verification. This approach to formal verification starts with the design of a formalization of the program one intends to prove by using a logic language. Then, a set of properties that the formalization must respect in order to be correct is provided to the tool, which attempts to verify them usually resorting to exterior help such as SMT solvers, for example.

2.2 Related Work

The effort that has been put into the verification of NNs over the years has resulted in significant scientific advance of the area. Even though there are still some major obstacles in the way, such as the scalability of the verification methods, it can be argued that the continuously developed work on the topic is going in the right direction in pursuit of NN's correctness, as work continues to be put into proposing methods [7], or the improvement of existing methods [8], that way expanding towards the different types of NNs. Software tools based on these methods have also been developed, three of them being presented in this section.

2.2.1 Reluplex

Reluplex [5] is an SMT Solver specifically designed to verify deep NNs that use the ReLU activation function. The necessity to develop an SMT Solver for this end and not using already existing ones comes from the difficulty in verifying NNs, caused by the presence of activation functions. To deal with this problem, Reluplex extends the simplex algorithm, which is a standard algorithm applied in solving linear programming instances, in order for it to be able to deal with the ReLU constraints. This is possible by taking advantage of the piecewise linear nature of ReLUs (Figure 3), by representing ReLUs as a set of disjunctions for example.

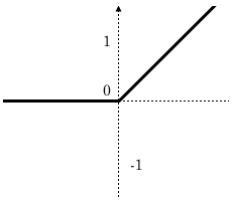


Figure 3: ReLU activation function.

[5] shows that Reluplex was tested on the prototype of the ACAS Xu system [9], the unmanned variant of the *Airborne Collision Avoidance System X (ACAS X)* that, as the name suggests, is a system made to avoid mid-air collisions between aircraft, focused on providing horizontal maneuver indications, which can be divided into *Strong Left*, *Weak Left*, *Strong Right* and *Weak Right*. This system maps these maneuver indications to variables, and the one assigned the lowest score corresponds to the best action to be taken at the moment. Therefore, the verification of this system focuses on checking if the networks assign correct values to the maneuver indications. For example, one of the verified properties was "If the intruder is near and approaching from the left, the network advises strong right". The representation of such property corresponds to the following constraints:

$$\begin{aligned} 250 \leq & \rho \leq 400 \\ 0.2 \leq & \theta \leq 0.4 \\ -3.141592 \leq & \psi \leq -3.141592 + 0.005 \\ 100 \leq & v_{own} \leq 400 \\ 0 \leq & v_{int} \leq 400 \end{aligned}$$

where ρ is the distance from the intruder to the own ship, θ is the angle to the intruder relatively to the direction the own ship is heading, ψ is the angle between the heading direction of the intruder and the own ship's heading direction, v_{own} is the speed of the own ship and v_{int} is the speed of the intruder (refer to [5] for more extensive information). The expected result for this property is for the value attributed to the "strong right" maneuver to be the minimal value.

2.2.2 Marabou

Marabou [10], a NN verification framework, is an SMT-based tool that implements the Reluplex algorithm while adding new and improved techniques and functionalities. Marabou verifies if NNs respect certain properties by representing those properties as a set of constraints, that are then tested to see if they are satisfiable by the network. It supports fully connected feed-forward and convolutional NNs, with varied configurations and piece-wise linear activation functions. This is one of the improvements over Reluplex, since Reluplex could only support ReLU functions.

The verification queries are composed of two parts, being one of them the NN to be verified, and the other one the property that will be checked. If the property is satisfiable, it will return SAT. If the property

proves not to be satisfiable, UNSAT is returned, along with a counterexample. Marabou is currently capable of answering two different types of queries:

- **reachability queries:** this type of query checks if, given an input in a certain range, the obtained output is in a desirable range that guarantees safety.
- **robustness queries:** it is known that NNs can be susceptible to attacks by recurring to the use of adversarial inputs [11], which are inputs that result from slight perturbations applied to inputs that were previously correctly classified by the network. This causes the network to classify them incorrectly. Robustness queries check if there are any adversarial points around the input received that may result in a change to the output of the network.

In order to reason about the query received, Marabou abstracts the network, representing each neuron as a variable with a value assigned to it. The values of these variables change in each iteration, in search of a variable assignment that satisfies both the linear constraints (the constraints that refer to the weighted sums) and the non-linear constraints (the constraints that refer to the activation functions).

With the well-known scalability problem that affects NN verification methods in mind, Marabou provides a *Divide-and-Conquer* mode that, as the name suggests, works by dividing the input region into sub-regions. This method then takes advantage of parallelism to deal with these sub-queries, by checking each one of them on a different node, which results in an improvement of Marabou's performance.

2.2.3 Verily

Reinforcement learning [12] is a sub-field of machine learning that focuses on a computational agent learning to automatically make decisions that give the best performance. This is achieved by following a trial and error process, where the impact of the chosen actions on performance is continuously evaluated.

Deep reinforcement learning is the result of combining reinforcement learning with NNs. In these systems, the NN is the responsible for learning how to make the decisions, taking as input the state of the environment when making the decision, and outputting the result of the decision.

To verify the correctness of these systems, *Verily* [13] was created, achieving its objective by performing bounded model checking and verifying if the required properties of the system can be satisfied. If the properties prove to be unsatisfiable, Verily provides a counterexample, that will help to understand in which scenarios the desired requirements are not being met. These properties can be divided in two groups:

- **safety properties:** a safety property asserts that nothing bad happens in the system.
- **liveness properties:** a liveness property asserts that something good will eventually happen in the system.

Verily makes use of Marabou as its engine, being Marabou the one behind solving the bounded model checking queries.

In one of the tests [13], Verily was applied to the verification of *DeepRM* [14], which is a system for managing cloud computing resources. This system keeps track of the usage of the resources (CPU and memory), a queue of jobs to be scheduled and the number of jobs beyond those already in the queue, being all this information the input to the NN. Various properties were tested, e.g., "When system resources are 0% utilized and there is a single large job in the queue, DeepRM's DNN always schedules that job" (refer to [13] for more extensive information). The results showed that this property in specific could be verified, however, some of the others could not, in which cases Verily provided counterexamples.

Chapter 3

The Why3 Proof Platform

In this third chapter of the present dissertation we present the Why3 Proof Platform, which is the tool used for the work here developed with NNs. Furthermore, we first provide a general overview of what this tool consists of and what it offers as functionalities. Then we proceed to exhibit three examples of Why3's usage, where one example focuses more on Why3's logical specification language and another focuses on WhyML, Why's programming language, while the third example presents something that does not quite fit any of those. all of the examples are explained, in order to provide a good understanding of how Why3 can be used in each case and, at the same time, to demonstrate the power of this tool.

3.1 Functionalities

Why3 [15] is a logical tool utilized for deductive program verification. It provides a rich logical language, equipped with inductive types and polymorphism, and a very intuitive programming language with a syntax and functionalities very close to those of many of the currently most used programming languages. Instead of being a theorem prover itself, Why3 can be seen as more of a framework that communicates with various theorem provers, which allows for greater flexibility in the sense that the user is not bound to just one prover that may not be capable of dealing with the given task.

Due to the fact that Why3's syntax is considerably different from the input syntax of the theorem provers, Why3's logic must undergo some changes before being ready to be passed as input to the provers. The ones in charge of this change process are the *drivers*, which are nothing more than configuration files, specifying the necessary transformations that allow Why3 and each theorem prover to interact. This modularity brings with it an interesting functionality, which is the possibility for the user to write its own transformations. This allows the user not only to change the interface of the supported provers, but to even add support for a new one as well.

Why3 comes with three main tools when installed, which may be used according to the user's needs:

- A simple command-line interface (CLI), which can be used to apply a certain prover in verifying the goals of a certain file, which must be written either in the Why3 language (.why extension) or in

WhyML (.mlw extension).

- A graphical user-interface, *why3ide*. In Figure 4, on the left side we can see the current status of the proof, where the green check mark icons show that those *goals* were successfully verified, whereas the clock icon shows that the prover timed out without being able to verify those goals and the blue question mark shows that those goals could not be verified, in which case different provers should be used or the logic corrected. Still on the left side, the provers being used at the moment are presented under each goal, which in this case are CVC4 [16] and Z3 [17]. There are also *splits*, that refer to the verification of the pre and post conditions. On the right, upper side, is where the content of the file is displayed and edited. Finally on the right, lower side, the system messages are displayed in case of errors, for example, along with the logs, the output of the prover and the counterexamples, if there are any.

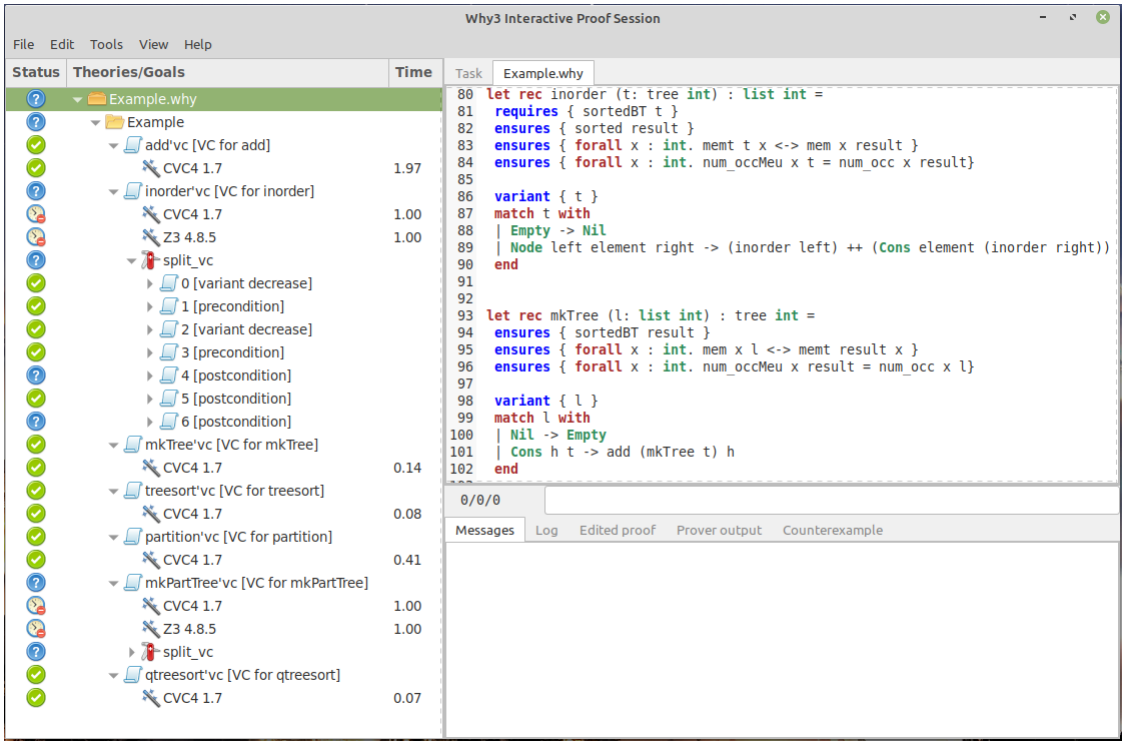


Figure 4: Why3ide, the Why3 graphical user-interface.

- A benchmarking tool, *why3bench*, used to benchmark different provers' performance, which can be very useful to compare different approaches on the same problem.

Why3 seems to be a promising tool concerning the formal verification of NNs for a number of reasons. For example, both languages featured by Why3 are very expressive, which may prove to be very important in abstracting and formalizing not only the NNs, but also the desired properties that these networks should respect. The fact that Why3, much like Marabou, resorts to, for example, SMT Solvers, is also a positive point worth mentioning. Lastly, the combination of different provers that Why3 uses will surely be a key point to this approach. And if none of the base provided provers appears to be suited for

the task at hand, the possibility that Why3 provides of adding support to different theorem provers may come to be important for the success of this study.

3.2 Why3 Utilization Examples

In the previous section it was mentioned that Why3 is equipped with two different languages, however this is not enough for the reader to understand how these can be applied and to grasp their richness when it comes to expressive power and to what these can do. For this reason, and with the ultimate goal of contributing to the existing documentation about Why3's usage which, contrarily to other better known formal verification tools, is not as abundant as we would like, we present below three chosen examples of formally verified programs from the *toccata* website[1]. These examples are here explained step by step, hoping to provide a better and easier to understand knowledge base for other possible users to learn how to use this powerful tool that is Why3 and maybe apply it to their own work, since learning by example is often the preferred method for many people. Considering that we are dealing with two different dimensions when explaining the two languages of Why3, being that one is purely logical and the other one is a programming language, the logical step was to have a specific example for each of these cases. The third example is a special case that does not fit any of those two sides.

3.2.1 The Towers of Hanoi

The first example to be analysed here is the programming language example, which handles the well-known classic problem of The Towers of Hanoi. For those that may be less familiarized with this problem, this is a mathematical problem designed by the french mathematician Édouard Lucas in 1883. As can be seen in Figure 5, this puzzle is comprised of three vertical rods and a number of disks n. These disks have different sizes and in the beginning of the puzzle, the state represented in the figure below, they are all in one of the rods on top of each other, sorted based on their diameter, from the largest one on the bottom to the smallest one on the top. The disks are then moved between rods, with the final objective of moving all the disks to the final rod. However, there are strict rules regarding these movements, which are as follows:

- The person solving the puzzle can only move one disk at a time.
- Only the top disk of any of the three rods can be moved.
- The moved disk must be placed on top of the other disks in the target rod, or on the empty rod if there are no other disks there.
- When there are already disks in the target rod, the disk being moved can only be placed there if it is smaller in diameter than the top disk in that rod.

[1] http://toccata.lri.fr/gallery/why3.en.html

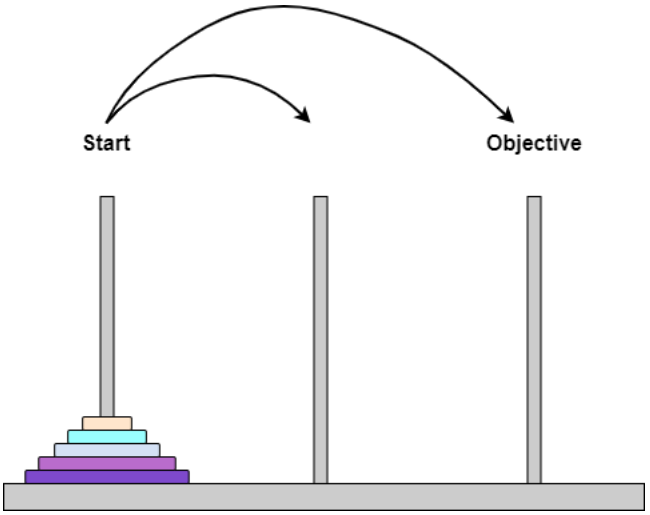


Figure 5: Towers of Hanoi puzzle.

3.2.1.1 Implementation

The implementation of this problem in Why3 is a recursive example. It represents the disks as natural numbers and the correct order of larger disk on the bottom and smaller on top is in this case assured by verifying that these numbers are in ascending order. The implementation starts with the declaration of the module, whose name should coincide with the name given to the file. It then imports the modules and theories from those modules that it will need which, in this case, are all provided by Why3, with the *use* keyword. For example, from the module *list* it will use the theories *List*, *Length* and *SortedInt*.

```
module Hanoi

  use int.Int
  use list.List
  use list.Length
  use list.SortedInt
```

After this, a custom type is set, a functionality that WhyML offers similar to the custom types that most programming languages also offer. In this case, this custom type has the objective of representing each one of the rods, and consists of a *mutable* list of integers, called *rod*. An invariant about this list is also defined, which expresses that the integer list *rod* must always be sorted in ascending order.

```
type tower = {
    mutable rod : list int;
  } invariant {
    sorted rod
  }
```

The *prepend* function is a recursive function from Why3's logical language, whose purpose is to append the integer n and all the integers between zero and n to the list s, outputting the final list of integers. For

example, if we had the case where the initial list was empty and n took the value of 3, then the result would be the following:

$$prepend\, n\, Nil \;=\; [1,\, 2,\, 3]$$

This function is not defined as normal functions are, instead its behaviour is defined by the two Axioms that can be found below it, one of them being the normal case where the value of n is greater than zero in which case n is appended to the beginning of the list and the function is called again now with $n \;=\; n-1$, and the other case is the stopping case, which prevents recursive functions from going on endlessly, and states that when the value of n is less than or equal to zero the list simply stays the same without any more calls. This function definition resorting to axioms is a perfect example of how the logical and programming languages can blend together to complement one another. Nevertheless, it is important to note that logical functions can also be defined as other functions are, which is very well represented by the predicate p defined in the next example, since predicates are nothing more than logical functions whose result is a *Boolean*.

```
function prepend (n: int) (s: list int) : list int

axiom prepend_def_zero :
  forall s: list int, n: int.
    n <= 0 -> prepend n s = s

axiom prepend_def_succ :
  forall s: list int, n: int.
    n > 0 -> prepend n s = prepend (n - 1) (Cons n s)
```

Then, the function *move*, responsible for moving a disk from tower a to tower b is defined. Note that in this function some of the arguments have the keyword *ghost* behind them. What this keyword does is setting those arguments as *ghost code*, which means that it is only used for verification and is not executed, and therefore does not influence the result. As can be seen, those ghost arguments are indeed only used in setting up the *contract* of this function. A contract is set using the keywords *requires* and *ensures*, and it is a contract in the sense that the function, given that its requirements are respected, will ensure those conditions inside the *ensures* statement are true at the end of each execution. In this function, the contract can be explained as follows:

- The first requirement means that the rod from where we will be taking the disk from must have some disk in it. Lists in Why3 are recursively defined, and are represented as

  $$type\; list\; a \;=\; Nil \mid Cons\; a\; (list\; a)$$

  which means that a list can either be empty, which is the *Nil* case, or it has one item of type a attached to a list of other items of type a, and that list can itself be empty or have items, and so on.

- The second requirement states that the rod to where the disk will be moved must either be empty or the top disk must be larger than the one that we are adding. Note that in this requirement, pattern matching is used, another very useful functionality that this language offers.

- The first condition to be ensured states that in the end of the execution of that function, the rod from which we removed the disk from now possesses the same disks except for the one removed.

- The second condition to be ensured, and last part of this function contract, states that after this function finishes its execution, the rod to where we moved the disk now possesses the same disks as before plus the new disk. Here we can see the usage of the keyword *old* that refers to the state previous to the execution of the function.

To fulfill its purpose, this function uses pattern matching with the keyword *match*, and here it is matching the possible states of the rod from which we remove the disk, which can have disks, in which case the disk is removed from the starting rod a and is added to the target rod b, or can be empty, which logically does not allow the execution of this function since the moving function needs a disk to move, and therefore this case is matched with the *absurd* keyword, meaning that if this case were to happen it would be logically incorrect.

```
let move (a b: tower) (ghost n: int) (ghost s: list int)
    requires { a.rod = Cons n s }
    requires { match b.rod with Nil -> true | Cons x _ -> x > n end }
    ensures  { a.rod = s }
    ensures  { b.rod = Cons n (old b.rod) }
  = match a.rod with
    | Cons x r ->
        a.rod <- r;
        b.rod <- Cons x b.rod
    | Nil -> absurd
end
```

Afterwards comes the function $hanoi\_rec$, responsible for implementing the known recursive algorithm used to solve this problem, resorting to the concepts and functions previously explained, only changing the order of the rods in each recursive call depending on which rod will receive the next disk to be moved. Note the use of the *begin* keyword in the first line of the implementation, used here to create a loop. This function, much like the previous one, also has a contract. However, as is presented below, this contract introduces a new concept, the concept of a variant. The contract is as follows:

- The first requirement, by using the $prepend$ function, states that in the beginning the starting rod has in it all the disks represented by the integers between zero and the current value of n.

- The second requirement serves the same purpose as the second requirement of the previous function, stating that the second rod must either be empty or its top disk must be larger than the next disk moving there.

- This third requirement is the same as the previous one, but now regarding the third and final rod.

- After the requirements, the concept of a variant is introduced. A variant is a value that must always decrease during the execution of the function, and is used to verify that the loop or recursion ends. In this case the chosen value is that of n which is easy to check if it decreases since in every recursive call the function is called with $n = n - 1$, decreasing until it reaches zero, when it stops.

- The first condition ensured states that, in the state after each execution, the starting rod now has the same disks except for the one that was moved.

- The second post-condition states that, after each execution, the target rod has the same disks plus the one that was moved and all those bigger than that one (these are moved during the other recursive calls, each of these calls calling with a smaller n, eventually leading to only one disk being moved at a time, respecting the rules).

- The last condition ensures that the auxiliary rod stays with the same disks as before the execution of the function.

```
let rec hanoi_rec (a b c: tower) (n: int) (ghost s: list int)
    requires { a.rod = prepend n s }
    requires { match b.rod with Nil -> true | Cons x _ -> x > n end }
    requires { match c.rod with Nil -> true | Cons x _ -> x > n end }
    variant  { n }
    ensures  { a.rod = s }
    ensures  { b.rod = prepend n (old b.rod) }
    ensures  { c.rod = old c.rod }
  = if n > 0 then begin
      let ghost t = c.rod in
      hanoi_rec a c b (n-1) (Cons n s);
      move a b n s;
      hanoi_rec c b a (n-1) t
end
```

Lastly, the $tower\_of\_hanoi$ function simply uses the previous function to do all the disk movements, providing the necessary arguments. To ensure the correct pre and post-conditions of this function, a contract is also used here as it was used in the two previous functions.

- The first requirement states that before the execution of this function, the starting rod has in it all the disks.
- The second requirement states that in the beginning the second and third rods have no disks.
- The first post-condition states that in the end the target rod has all the disks.
- The second post-condition states that in the end both the starting rod and the auxiliary rod have no disks.

```
let tower_of_hanoi (a b c: tower)
    requires { a.rod = prepend (length a.rod) Nil }
    requires { b.rod = c.rod = Nil }
    ensures  { b.rod = old a.rod }
    ensures  { a.rod = c.rod = Nil }
  = hanoi_rec a b c (length a.rod) Nil
```

Lastly, to close the module, the *end* keyword is used. Note that the use of that same keyword in the functions $move$ and $hanoi\_rec$ do not serve the purpose of terminating the functions as that is not necessary. Instead, these are terminating the pattern matching statement and closing the loop, respectively.

3.2.2 Mathematical Puzzle from a Dijkstra Paper

For the second example of the utilization of Why3, from the same source, we chose a formalization of a small mathematical puzzle from a paper written in 1995 by the renowned computer scientist Edsger Dijkstra. In this example, the problem is formalized solely using the logic language provided by Why3. This problem aims to prove that

$$\forall_n \ : \ f(n) = n$$

considering a function f over the set of natural numbers $\mathbb{N}$, such that

$$\forall_n \ : \ f(f(n)) < f(n+1)$$

3.2.2.1 Implementation

To solve this problem in Why3, the formalization was divided into three theories. A theory is a group of declarations like functions, axioms, lemmas, goals and so on, and can be imported into other theories to help proving their goals.

The first theory is called Puzzle, and it is responsible for setting up the problem. The formalization starts exactly by setting the name of the theory, using the *theory* keyword. Then another theory is imported to be used in this proof, the theory *Int* from the module *int*. We can see here that instead of just using the *use* keyword, the *export* keyword is also added, meaning that besides importing the content from that theory, if this theory that we are presenting is later imported somewhere else with the name P, then its

content would be accessible as $P.name$, where name can be a function, a lemma, or any other part of the theory.

```
theory Puzzle

  use export int.Int
```

After this initial setup, the function f that is mentioned in the problem is declared as being a function that works with integers. Afterwards, two Axioms are declared, the first one, called $H1$, stating that for all n that is an integer, if zero is less than or equal to n, then saying that zero is less than or equal to $f(n)$ will also be true, which can be represented as:

$$\forall_{n \in \mathbb{N}} \ : \ 0 \leq n \Rightarrow 0 \leq f(n)$$

The second Axiom, $H2$, states that for all n that is an integer, if zero is less than or equal to n, then $f(f(n))$ is less than $f(n+1)$, mathematically represented as:

$$\forall_{n \in \mathbb{N}} \ : \ 0 \leq n \Rightarrow f(f(n)) \leq f(n+1)$$

The function f and these two Axioms, when implemented in Why3 take the following form:

```
function f int: int

axiom H1: forall n: int. 0 <= n -> 0 <= f n
axiom H2: forall n: int. 0 <= n -> f (f n) < f (n+1)
```

After these declarations, the present theory is closed with the *end* keyword, the same way as with modules. The second theory is called Step1 and has the objective of verifying the following, resorting to induction over k.

$$k \leq f(n+k)$$

As is the case of every theory, this one starts by declaring its name, following with the import of the Puzzle theory previously defined. Subsequently, a predicate p is defined, declaring that for all n that is an integer, zero being less than or equal to n implies that k is less than or equal to $f(n+k)$, which is to be proved by induction over k as was previously mentioned. To achieve this goal, the theory *SimpleInduction* is cloned, a theory used to prove simple inductions, and what cloning does can be described as copying and pasting that theory into this present one. When cloning, the predicate p is indicated as the predicate that is to be proved, using the lemma $base$ and the lemma called $induction\_step$, being then declared the end of this theory.

```
theory Step1

  use Puzzle

  predicate p (k: int) = forall n: int. 0 <= n -> k <= f (n+k)
  clone int.SimpleInduction as I1
    with predicate p = p, lemma base, lemma induction_step
end
```

The third and final theory in this formalization is called Solution and, as the name suggests, this is the theory where the previous two are used and the problem is solved. Following the declaration of the theory, the other two theories, Puzzle and Step1, are both imported so what was defined in them can be used here. Ensuing these imports, two lemmas are defined, where the first one, $L3$, declares that for all n that is an integer, zero being less than or equal to n implies that n is less than or equal to $f(n)$ and that $f(n)$ is less than or equal to $f(f(n))$, as is represented below:

$$\forall_{n \in \mathbb{N}} : 0 \leq n \Rightarrow n \leq f(n) \wedge f(n) \leq f(f(n))$$

The second lemma, $L4$, states that if zero is less than or equal to n, then that means that $f(n)$ is less than $f(n+1)$, as follows:

$$\forall_{n \in \mathbb{N}} : 0 \leq n \Rightarrow f(n) < f(n+1)$$

and both of these lemmas are defined in the Why3 formalization as can be seen below.

```
lemma L3: forall n: int. 0 <= n -> n <= f n && f n <= f (f n)
lemma L4: forall n: int. 0 <= n -> f n < f (n+1)
```

Following the two aforementioned lemmas, another predicate p' is defined over an integer k, stating that for all n and m, being both of them integers, if $0 \leq n \leq m \leq k$ then $f(n)$ is less than or equal to $f(m)$. This predicate is given the same usage as the predicate p previously examined, being used to prove by induction that f is increasing, resorting again to the theory *SimpleInduction* just like in the Step1 theory, as denoted below:

```
predicate p' (k: int) = forall n m: int. 0 <= n <= m <= k -> f n <= f m
clone int.SimpleInduction as I2
  with predicate p = p', lemma base, lemma induction_step
```

Finally, two more lemmas are defined, further closing in on the final objective to be proven that was initially described in the problem. Then a goal is defined, which represents the hypothesis being verified in this problem. The first of these two lemmas, $L5$, states that for all n and m, both of them integers,

if the value of n is between zero and the value of m, then this means that $f(n)$ is less than or equal to $f(m)$.

$$\forall_{n,m \in \mathbb{N}} \ : \ 0 \leq n \leq m \Rightarrow \ f(n) \leq f(m)$$

The second of these lemmas, and last lemma of this example, is the step exactly before proving the goal, and declares that for all n that is an integer, if zero is less than or equal to this n, then $f(n)$ is less then $n + 1$.

$$\forall_{n \in \mathbb{N}} \ : \ 0 \leq n \Rightarrow \ f(n) < n + 1$$

Finally, the goal tackles the initial objective of the problem and aims to verify that for all integers n, if zero is less than or equal to this n, then this implies that $f(n)$ is equal to n.

```
lemma L5: forall n m: int. 0 <= n <= m -> f n <= f m
lemma L6: forall n: int. 0 <= n -> f n < n+1

goal G: forall n: int. 0 <= n -> f n = n
```

3.2.3 The Area of a Triangle

This example is not a very complex one, since it simply consists of the code to calculate the area of a triangle. However, it is a very interesting example, due to the fact that it presents another functionality provided by Why3 in the field of languages, without being itself a language. Besides offering its logical language and the programming language, Why3 also offers the possibility of providing it with code written in C language or in Python with specific annotations in comment form, that allow the user to specify pre-conditions and post-conditions about this code, as was demonstrated in the first example with the *requires* and *ensures* keywords. These conditions are then verified by Why3 in order to prove the correctness of this code. In order for them to be recognized by Why3 they must be inside comments and be preceded by the *at* symbol. The following example of the code used to calculate the area of a triangle given its side lengths is a perfect example of this.

```
    /*@ requires 0 <= x;
  @ ensures \result==\round_double(\NearestEven,\sqrt(x));
  @*/
double sqrt(double x);

/*@ logic real S(real a, real b, real c) =
  @  \let s = (a+b+c)/2;
  @        \sqrt(s*(s-a)*(s-b)*(s-c));
  @ */
```

```
/*@ requires 0 <= c <= b <= a && a <= b + c && a <= 0x1p255;
  @ ensures 0x1p-513 < \result
  @   ==>\abs(\result-S(a,b,c))<=(53./8*0x1p-53+29*0x1p-106)*S(a,b,c);
  @ */

double triangle (double a,double b, double c) {
  return (0x1p-2*sqrt((a+(b+c))*(a+(b-c))*(c+(a-b))*(c-(a-b))));
}
```

Notice that the second annotation is defining a function S, and this function is then used as an auxiliary function in the annotation of the $triangle$ function below.

Chapter 4

Preliminary Work and Results

This third chapter goes over the preliminary work developed during the initial stages of this dissertation. Starting by explaining the decision of creating a proof-of-concept of a formalization of a NN, we then expose the reasoning behind every aspect of this formalization, from the representation of the neurons to the properties that were considered to be applicable to the represented NN. Finally, the results of this preliminary experience are displayed, while also explaining how these results influence the main project of formalizing a real-application-scale NN.

4.1 Preliminary Work

With the objective of better understanding how we could represent a NN using the language that Why3 offers and, most important of all, to see if it could be done, after the initial theoretical research, the investigation took a more practical approach, focusing on the creation of a proof-of-concept.

4.1.1 Neural Network Model

For this proof-of-concept we opted to represent a very simple NN, represented in Figure 6. This choice was based on the hypothesis that this small network would allow us to verify if this formalization could be done since it is still a NN and as such the concepts applied on its representation could be carried over to greater scale networks. Furthermore, this simple network would not be very demanding for Why3 in terms of solving and computational power, since dealing with that question was not the focus at this still primitive point of the study.

The network chosen for this purpose was a very simple instance representing the OR logical operation, a simple perceptron consisting of two input nodes, $x1$ and $x2$, and one output node, $y1$. The two input nodes would receive as input values either *True* or *False*, here represented with 1 and 0, respectively. The expected result for each combination of input values can be seen in Table 1.

x1	x2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Table 1: Logical-OR operation values.

For this simplified version, the bias value was disregarded, since in terms of complexity it would only bring one more integer to be added to the weighted sum when calculating the value of the neuron. The activation function here is represented by a simple threshold value set at 1, and the weight of both edges is set to 1.1 . If the result of the weighted sum is greater than or equal to 1, then the value of the output is set to *True* (1), otherwise it is set to *False* (0).

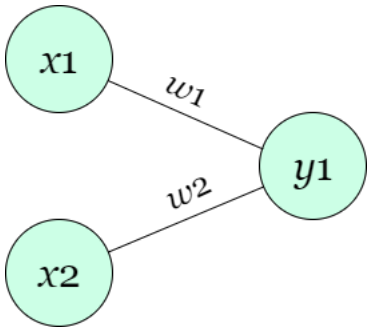


Figure 6: Model of the Neural Network used for the proof-of-concept.

4.1.2 Formalization of the Neural Network

When considering how to represent a NN using the language provided by Why3, it is first necessary to look upon NNs from a superior abstraction level, since Why3 does not possess any specific method to represent these networks due to the fact that it was not created with that objective in mind. The structure and syntax of this language, closer to the structure of normal programming languages than other formal verification languages, like Coq[18] for example, do offer the necessary support for us to abstract the concept and its properties, and therefore we are able to write a function that correctly depicts the functioning of a NN.

From this higher level of abstraction, it is not difficult to come to the conclusion that, mathematically speaking, the neurons composing the network can easily be represented by variables whose value depends on the values of all the variables that represent the neurons from the previous layer, the same path followed by Marabou. Thus, the expression of each variable composing the formalization of this perceptron takes

the form of the following mathematical expression:

$$N_{x,y} = f(\sum_{y=1}^{k} N_{x-1,y} * w_{x-1,y})$$

where $N_{x,y}$ is the y^{th} neuron in layer number x, k is the total number of neurons in the previous layer, and $N_{x-1,y}$ and $w_{x-1,y}$ represent the y^{th} neuron from the previous layer and its correspondent weight, respectively. An exception to this is the input layer, whose neurons are represented by the arguments received by the function for purposes of simplifying both the function itself and possible properties regarding that function that are necessary to prove its correctness. However, since there is only one neuron to formalize, considering that the input nodes are the arguments of the function, the form taken by this network when formalized in Why3 is that of a classic if-then-else statement, which includes the threshold.

$$if\ 1.1 * input1 + 1.1 * input2 >= 1.0\ then\ 1.0\ else\ 0.0$$

4.2 Verification of the Network and its Properties

The first step in proving the correctness of the network was directed towards validating its coherence, i.e., to check that the provided formal representation was logically and mathematically sound. Why3 is able to check this without the help of exterior provers, and will point any error found until the formalization is valid. Since Why3 did not indicate any existing error within our model, this showed that it was correct in the aforementioned terms.

After the coherence of the model was proved, the focus was directed towards the correct functioning of the model, as being logically and mathematically correct does not imply that the network executes the desired operation as it should. To that end two properties that the network should obey were formalized, along with a constraint for the possible input values.

The constraint applied to the input, here representing the first layer of neurons of the network, made sure that the only possible value that any of the neurons of that layer could assume was either 1 or 0, since we were dealing with a logical operation. This can be achieved with the *requires* keyword in Why3, specifying a pre-condition. This pre-condition took the form shown below:

$$requires\ \{\ (input1 = 0.0 \vee input1 = 1.0) \wedge (input2 = 0.0 \vee input2 = 1.0)\ \}$$

The two formalized properties fall into the category of reachability queries, since these follow the form *"given the inputs X and Y, the output should be Z"*. These properties had the objective of checking if the logical-OR operation executed by the network was producing the expected outputs. As can be seen in Table 1, there are four possible results. However, these four can be divided into two groups, further simplifying this formalization: if at least one of the two input nodes takes the *True*(1) value, the expected output is also *True*(1), and if both nodes take the *False*(0) value, then the expected output is also *False*(0). This

simplification left us with two properties to check, whose representation can be achieved with the *ensures* keyword, as follows:

$$ensures\ \{\ input1 = 1.0 \vee input2 = 1.0 \Rightarrow result = 1.0\ \}$$

$$ensures\ \{\ input1 = 0.0 \wedge input2 = 0.0 \Rightarrow result = 0.0\ \}$$

Unlike the case where the coherence of the model was checked, in this case Why3 calls exterior provers to deal with the task of verifying if the given properties are valid or not. In this case, the provers we used were Alt-Ergo version 2.4.1, Z3 version 4.8.6 and CVC4 version 1.7, all of them well known SMT Solvers. All the three of them were able to prove both properties as valid, with Alt-ergo performing considerably better both in terms of execution time and necessary steps, as can be seen in Table 2. Due to the fact that the properties are very similar in terms of validation process, the results were the same for both properties.

	Time (s)	**Steps**
Alt-Ergo	0.01	4
CVC4	0.07	2112
Z3	0.06	6701

Table 2: Execution times and number of steps taken by each solver when validating both properties.

4.3 Discussion

These initial results were important in setting the tone for the next steps in this research. The obtained results can be divided in two parts, and each of them has a different degree of importance considering their implications for the work presented in the next chapter.

The first part of the results, the fact that we were capable of formalizing the network using WhyML and Why3 successfully verified its coherence is with no doubt the most important part of our initial results, since this was in great part the main objective of this study, and therefore was very favourable to our hypothesis. However, despite this being a very positive outcome for the preliminary tests, it could not be regarded as more than a successful proof-of-concept, since the aim of this research was set at formalizing a real life application NN, and our simple perceptron did not possess the necessary scale or complexity to that end. Nonetheless, this proof-of-concept did provide the base principles regarding how to formalize a NN using Why3, and this could be extrapolated for greater scale networks, as is discussed in the next chapter.

The second part of the results refers to the two properties that were proven. Whilst the validation of these two properties was successful in this specific case, this did not imply that the same applies to the properties we attempted to prove regarding the network used in the work discussed in the next chapter. This is due to the fact that, what makes the verification of NNs such a difficult problem are the non-linear activation functions, which make the problem non-convex. In this initial prototype, the place of the

activation function is occupied by a simple threshold which, allied with the small scale of the network, offers little difficulty to Why3 or to the solvers used.

When dealing with a real application scale network, however, the situation changes. These types of networks come in a considerably larger scale which, by itself, constitutes a possible obstacle in terms of required computation power and time to validate. Adding to that, the solvers have to deal with the greatest obstacle, since these networks possess an actual activation function.

Chapter 5

Results and Findings

The present chapter scrutinizes the final results of this dissertation and discusses how these contribute to support our initial hypothesis of utilizing Why3 in the formalization of NNs. It starts by explaining every aspect of the formalization and how much of it was influenced by the initial work discussed in the previous chapter, then passing on to briefly describe the developed auxiliary generation script. Afterwards, the properties intended for verification, along with the respective results, are presented and, finally, the obtained results are discussed.

5.1 Formalization of the Neural Network

After the development of the previously discussed proof-of-concept, a greater scale network was necessary, one that matched state-of-art networks, as our objective was its formalization using Why3. To that end, we used one of the 45 NNs that compose a prototype of the ACAS Xu system, earlier presented in the second chapter of this dissertation, the same system used to test Reluplex. Each one of these networks has 8 layers with a total of 300 neurons, and use the ReLU activation function, which makes these a very good fit for the scale and complexity requirement that we were aiming for when searching for candidate networks.

When developing a formalization to represent this network, we mostly followed the base concepts acquired during the creation of the proof-of-concept, such as depicting every node as a variable, again with the exception of the input nodes, whose values are passed as input to the function. However, some things changed when comparing the proof-of-concept with the formalization of a NN of this scale and complexity. When representing this network, its scale did not permit the same simplification that was made in the simpler network, that allowed the function that represents it to be composed just by a simple if-then-else statement that already included the threshold verification. Nonetheless, the base concept of representing every node as a variable remains unchanged, and was applied here.

As mentioned, this network has an activation function that must be included when calculating the value of each node and, due to that fact, needs to be formalized. Activation functions are simple mathematical functions and pose no obstacle to WhyML in terms of representation. The ReLU activation function can

easily be implemented as an if-then-else statement, which could be integrated into the declaration of each variable. However, had we gone down this path, we would have been adding unnecessary complexity to the formalization, which should be as easy to understand and intuitive as possible. With this in mind, we took the decision of representing the activation function as a separated function. This function receives as input the value that it is supposed to standardize, and its inner workings can be coded as an if-then-else as follows:

$$if\ value > 0.0\ then\ value\ else\ 0.0$$

Being represented as a function itself, the ReLU activation function can now be called as any other function in most programming languages, thanks to the syntax that WhyML provides. Thus, we call this function on the result of the calculation of the value of each neuron, an approach that should be very familiar and intuitive for anyone working with software development. For example, this is how we represent the value of a neuron of the second layer of this network:

$$relu((w1 * input1) + (w2 * input2) + (w3 * input3) + (w4 * input4) + (w5 * input5) + bias)$$

An exception to this is the specific case of the output layer. When formalizing a NN, we want to be able to access the values being outputted by the final layer, so we can verify that it is producing the expected results. WhyML allows us to access the output of a function by using the *result* keyword, which is very important when verifying properties about the network's functioning. However, if we had followed the same formalization template and represented the output neurons as every other in the network, the *result* keyword would only give us access to the value of the last neuron. To deal with this problem, we decided to represent the last layer as a tuple constituted by the expression representing each of the output nodes, due to the fact that WhyML offers a very simple way to access the values contained in a tuple, as is demonstrated in the Properties section within this same chapter. With this solution, the *result* keyword gave us access to the whole tuple, from where we could get the value of every node, allowing us to represent any necessary reachability queries.

5.2 Auxiliary Generation Script

The idea of representing each node of a network as a variable is a good idea and works very well. However, it has a small drawback in what can be designated as the quality-of-life section. When we are dealing with a large NN, containing several hundreds of neurons, writing the expression of every single one of them by hand does not seem very practical or time efficient. To deal with this issue, we developed a simple Python script that aims to automate the generation of the expressions representing the neurons.

The NNs from the ACAS Xu system come in a specific file type, with the .nnet file extension. However, it is important to notice that this file format has no correlation with other frameworks that may produce files with the same extension. This file format was initially designed to represent aircraft collision avoidance NNs in a human-readable way, and can be found on github[1], along with all the necessary documentation

about it. Since our case study was a NN from the ACAS Xu system, our script deals only with networks in nnet files, receiving these as input.

Initially, in order to parse the file, we used a function provided in the github repository, with just a minor change for the function to return the weights and the biases, since those are the values we need from the network to generate the expressions. We then defined a function with the purpose of generating the expressions, both those of the "normal" nodes and the expression representing the output layer. This function writes these expressions to a Why3 file that is ready to be read by Why3, however, before trying to verify anything, we must first write ourselves the signature of the function since the generator does not create that.

5.3 Verification of the Network and its Properties

Following the same organization set in the last chapter, this section is divided into the results of the verification of the network's consistency and the verification of properties about its functioning. After completing the formalization of the network and the activation function, Why3 was able to validate that it was consistent, just like it had done with the proof-of-concept discussed in the previous chapter.

Having successfully verified that our formalization of the network and activation function were both consistent, it was time to direct our attention towards the properties and constraints imposed upon this model. These properties and constraints were not designed by us, but rather by the team behind Marabou that kindly provided them after we established contact with them, and our job here was to write these properties in WhyML in order to see if the provers supported by Why3 could successfully verify them.

To begin with, we present the constraints imposed upon the input, to limit the allowed range of values. Since the input of this network is comprised of sensor data regarding the aircraft's speed and current course and sensor data related to any other aircraft in the vicinity, these constraints are setting the expected input value range received by the NN in accordance with the sensors' output ranges. When writing these constraints on Why3 we incorporated all of them in the properties, since the six properties about this network are reachability queries. When writing this type of property in WhyML, we always have the form *"input X implies output Y"*, therefore if we had also represented the input constraints with the *requires* keyword like in the proof-of-concept, we would have redundancy in our formalization. However, it is important to note that in the proof-of-concept this is not the case and there is no redundancy observed, since the input constraints are setting the possible values that the input values can take and then for the properties we have very specific values for the input nodes. In the present case, the values of the input nodes set in the constraints are the same as those in the properties, and as such we can simply represent the input constraints only inside the properties. Despite this, and for the purpose of offering better readability, we first expose them here. The input constraints are:

$$input1 \leq 0.679858 \wedge input1 \geq (-0.328423)$$

[1] https://github.com/sisl/NNet

$$input2 \leq 0.5 \land input2 \geq (-0.5)$$

$$input3 \leq 0.5 \land input3 \geq (-0.5)$$

$$input4 \leq 0.5 \land input4 \geq (-0.5)$$

$$input5 \leq 0.5 \land input5 \geq (-0.5)$$

The five properties about this network have the objective of verifying that it will output the correct values which, given the context of the ACAS Xu system, translates to the aircraft receiving the correct instructions that will allow it to avoid any possible collisions with other aircraft. As was mentioned in the previous chapter, in order to verify this type of query we needed to access the values of the output neurons. Given that we represented the whole output layer as a tuple, and that WhyML allows us to easily access both the values of a tuple and the result of the function, we follow with an example of a property about this network that exhibits these functionalities offered by WhyML, where the input constraints are not represented in full since they were already presented above. As is the case of the properties in the proof-of-concept, to code these in WhyML we make use of the *ensures* keyword, as can be seen bellow:

$$ensures\ \{\ let\ (out1, \_, \_, \_, \_) = result\ in\ (input\ constraints) \Rightarrow out1 \geq 0.52\}$$

Here we can see that we use *out1* to access the value of the first neuron in the output layer while we ignore the other output values, since the property we are trying to verify only deals with the first, allowing us to use its value in our validation.

To deal with the task of verifying the correctness of these properties we used a varied selection of provers supported by Why3, namely Alt-Ergo version 2.4.1, Z3 version 4.8.6, CVC4 version 1.7, Vampire version 4.5.1 and E prover version 2.6. However, even though we used several well-established provers, none of them was able to verify any of the properties, all of them having timed out in every attempt.

5.4 Discussion

As was done in the Discussion section of the previous chapter, in this section we separate the obtained results in two categories, being the first one the results related to formalizing the real life application scale NN, and the second one the results regarding the verification of its properties in order to verify its correct functioning.

The first part was a success. As we had theorized, we were able to extrapolate our formalization method used in the proof-of-concept into the case of a much bigger NN, applying the same concepts with less expression simplification. Why3 also proved capable of validating the consistency of this network. Even though it is of a much greater scale than the previous one, Why3 showed no signals of struggle when dealing with this problem.

The second part is where the difficulties revealed themselves, although this was not unexpected. As we had mentioned in the homologous section in the previous chapter, the fact that the solvers used were able

to verify the properties about the perceptron did not imply that the same would happen when dealing with a Network possessing an actual activation function, due to the aforementioned reasons. This, however, can be easily corrected, as the problem does not reside in Why3 itself, but in the capabilities of the solvers to deal with this. As was explained in Chapter 2, one of the pros that Why3 has is that even if all its supported provers fail, it offers the functionality for the user to add support to new provers.

Chapter 6

Conclusions and Future Prospects

In the beginning of this research, the goal was set at trying to formalize a NN using Why3, to see if it could be done and if Why3 was capable enough to be a plausible formal verification tool regarding NNs. This, to the best of our knowledge, had never been done before and as such we had an innovative hypothesis that, proven correct, could add a new tool to the repertoire of already existing tools developed to this end. Our first experiments yielded a very favourable outcome as was shown in Chapter 3, as these results were not just successful in the sense that we managed to formalize a perceptron and verify two properties about its functioning, but mostly because these positive results allowed us to build a solid understanding with respect to how to formalize a NN using WhyML, the language provided by Why3. However, our objective was to formalize a real life application scale NN, since the small scale and complexity of a perceptron could not be compared with the current state-of-art NNs being used in the applications we see everyday.

We then proceeded to work with a proper NN, resorting to the ACAS Xu system, which did pose some obstacles to Why3. This network was of a considerably greater scale with its 300 neurons, when compared with our 3 neuron perceptron. The theory established during the development of the proof-of-concept about extrapolating the representation principles from the perceptron to greater scale networks was shown to be correct and, using those same principles, we were able to formalize this network, with the help of a generator script used to generate the expression of the nodes. Even with this considerable scale, Why3 did manage to verify the consistency of the network since it did not need the help of solvers for that task, which came as an important success. Formalizing the activation function was simple work, as it was simply a mathematical function and Why3 has much more than the necessary arsenal for that matter. It is also very important to note that, not only did Why3 perform very well in terms of formalizing these two components, but it did this in an extremely intuitive way to anyone used to work with writing code thanks to the structure of its language, which is something that many formal verification tools fail to achieve.

The difficulties presented themselves when we tried to push Why3 beyond our initial objective of formalizing a NN. Since we were able to achieve this goal, attempting to verify properties about this network was the next logical step. Despite the fact that we managed to verify properties on our proof-of-concept,

we were completely aware that this had no implications towards any properties on this network, as the case of the perceptron was a very undemanding one where we were dealing with a simple threshold. The situation with the ACAS Xu network was very different, since this one did possess an activation function, the Achilles' heel of NNs formal verification. Formalizing the properties was as simple as formalizing the ones related to the perceptron, however, verifying them was not possible, since none of the solvers used was able to output any other result than a timeout. In spite of that, this problem can be solved, resorting to one of the functionalities offered by Why3, which allows the user to add support to new solvers.

In conclusion, we do consider this research to be an overall success, considering that the main objective was achieved, and even though the extra self-imposed goal was not successful, this was an issue with the solvers and not with Why3 itself, and does not constitute a final barrier for the aforementioned reason. When it comes to the formal verification of NNs, we now know that Why3 does have a lot of potential, from its very expressive and rich logical language to the fact that it allows the user to add new provers if necessary, being that the first was invaluable to this dissertation and the second will be extremely important for the work planned for the future of this research.

The future work that we here propose will surely bring Why3 forward as a new formal verification tool for NNs. As was discussed above, we have shown in this dissertation that Why3 and WhyML are fully capable of formalizing a state-of-art NN and verifying its coherence, only failing at proving its properties due to the fact that the supported provers were not able to solve them. The first future objective means to directly tackle this problem, and so we must first and foremost work on adding support to a solver that is able to deal with these properties, being Reluplex the best candidate for this place since it was specifically designed with this purpose in mind, and this addition should finally establish Why3 as a NN formal verification tool. In second come some necessary improvements to our existing solution. As it is right now, the size of our formalization in terms of lines of code is directly proportional to the amount of neurons in the network being verified, which can quickly get out of hand. And while we did provide the auxiliary script to help with this issue, we are sure that Why3 offers a way of representing this in a more optimized way, perhaps resorting to a loop, a control flow statement that WhyML provides and that is bound to be familiar to most people who possess some form of programming knowledge. If for some yet unknown reason this is not possible, then the script will still be necessary for the quality-of-life improvements that it offers. However, this too can be improved. Starting with making the script also generate the signature of the function, receiving the name of the function as an argument to the script, we can then add the capability to generate the formalization of the activation function, provided that the user indicates the function being used.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] Rasool Fakoor, Faisal Ladhak, Azade Nazi, and Manfred Huber. Using deep learning to enhance cancer diagnosis and classification. In *Proceedings of the international conference on machine learning*, volume 28. ACM, New York, USA, 2013.

[3] Kuniaki Noda, Yuki Yamaguchi, Kazuhiro Nakadai, Hiroshi G Okuno, and Tetsuya Ogata. Audio-visual speech recognition using deep learning. *Applied Intelligence*, 42(4):722–737, 2015.

[4] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[5] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[6] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, Oct 1986. ISSN 1476-4687. doi: 10.1038/323533a0.

[7] Yuval Jacoby, Clark Barrett, and Guy Katz. Verifying recurrent neural networks using invariant inference. In *International Symposium on Automated Technology for Verification and Analysis*, pages 57–74. Springer, 2020.

[8] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T Johnson. Improved geometric path enumeration for verifying relu neural networks. In *International Conference on Computer Aided Verification*, pages 66–96. Springer, 2020.

[9] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2016.

[10] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[11] Alexey Kurakin, Ian Goodfellow, Samy Bengio, et al. Adversarial examples in the physical world, 2016.

[12] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[13] Yafim Kazak, Clark Barrett, Guy Katz, and Michael Schapira. Verifying deep-rl-driven systems. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 83–89, 2019.

[14] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

[15] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[16] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.