Chapter 1

A Novel System Design and Implementation

In this Chapter, we present in detail the methodologies and technologies proposed and implemented in this thesis.

The motivation of this thesis was to introduce a more domain-specific approach to the task of neural network verification and parameter optimisation by employing the use of external neural network verifiers in the SMLP workflow.

As discussed in Chapter 1.1, it was expected that, by employing a state-of-the-art external neural network verifier, the verification process should be handled more concretely. This means that the verification process should use more advanced methods than the current implementation of verifying the neural network, which consists of encoding the neural network's weights and biases and feeding them into an SMT solver. It should also allow for more complex deep neural networks to be verified since Marabou supports many architectures, including most of the non-linear activation functions.

Chapter 1.2 is devoted to the PySMT format. Before, the SMLP system employed the SMT-LIB encoding to create and describe the formulas that act as inputs to the Z3 solver. Considering that the external neural network verifiers are not SMT solvers, *in this thesis we propose a novel approach that adapts an alternative format to translate the input formulas*. Specifically, we have chosen the PySMT format, which we have translated the formulas into. Part of the proposed architecture's workflow that we have implemented for the purpose of this thesis, focused on the translation of the formulas that are represented in a symbolic format in the code (through the Python "operators" library) and will, in the future, act as constraints to the neural network.

In Chapter 1.3, we show that for the purpose of this thesis, we determined on

nominating Marabou as the principal external neural network verifier that was natively integrated into the system. We also illustrate that we were in the position to directly produce and communicate their inputs and parse the results of the neural network verification query.

Finally, in Chapter 1.4, we present the architecture of this integration and the integral components of which it is comprised of. We also discuss the introduction of additional interfaces that serve the purpose of triggering the correct solver implementation during runtime. Furthermore, certain use cases that required customised approaches during different process phases, like formulas generations, equations and inequalities parsing into Marabou constraints and mitigating Marabou API limitations to curated solutions, are thoroughly discussed.

Fig. 1.1 depicts the flow of data alongside the use cases and procedures that are executed during the PySMT constraints generation, Marabou constraints representation and Neural Network Verification pipelines. Ultimately, we use the results extracted from the Marabou solving procedure, process the solution and return it to the main SMLP workflow.

1.1 Describing the existing architecture

1.1.1 Specification File

The inputs into the neural network have semantic differences and are split into two categories, inputs and knobs. Their primary distinction is that knobs are decided in the design phase prior to the system's execution whilst input values remain free and are decided during runtime execution [BKK24]. In the formal definition of model exploration tasks, knobs and inputs correspond to existentially quantified and universally quantified variables, respectively. Thus, in the typical contexts of verification, optimisation, and synthesis:

- For verification, all variables are inputs.
- For optimisation, all variables are knobs.
- For synthesis, some variables are knobs and the rest are inputs.

The constraints focus on specific neural network variables and are defined in a specification file, which serves as an input to the SMLP system. These constraints include:

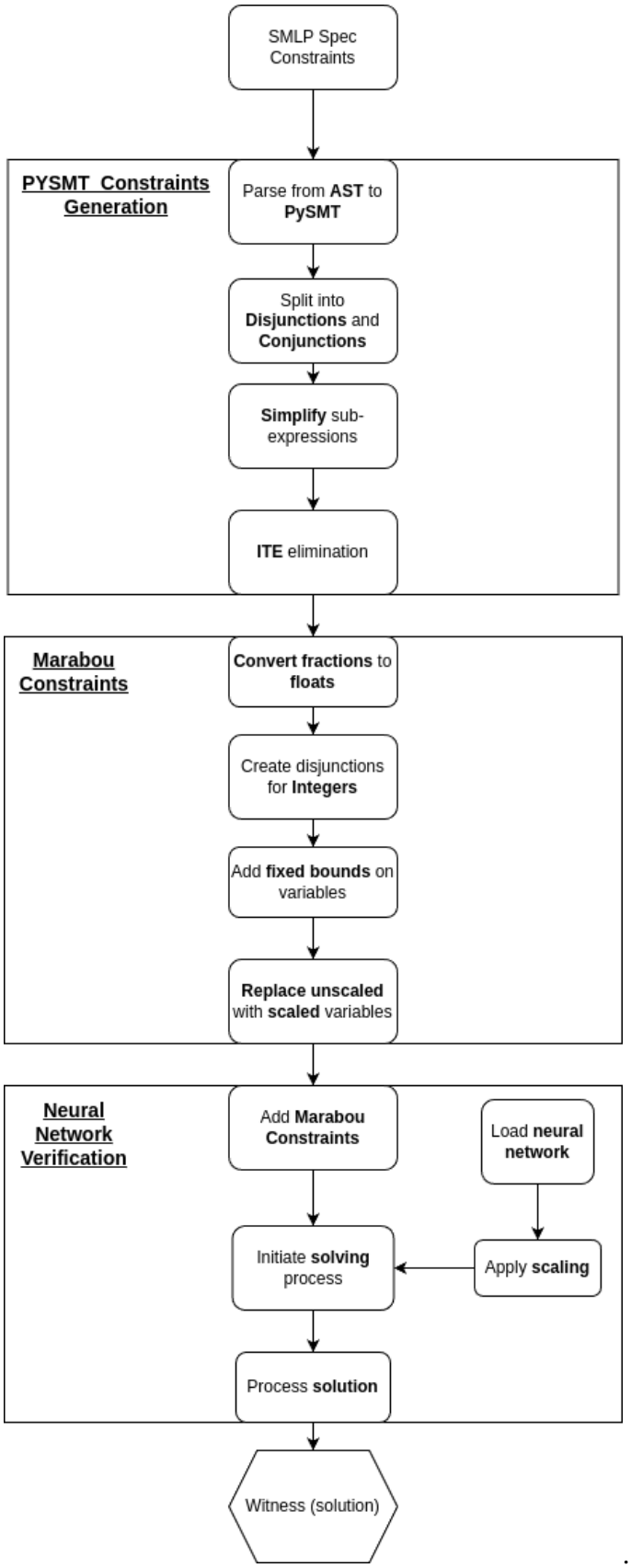

Figure 1.1: SMLP workflow

- variable constraints: Each variable, when declared in the specifications file is accompanied by its: I/O type (Input, Output), value type (Integer, Real), the allowed value range and lastly, if the variable is an integer, a grid of permissible values.
- alpha (α): Formula that contains further restrictions on the input and knob values.
- beta (β): Formula that constrains the output variables.
- eta (η): Formula that contains knob input ranges and extra constraints.

Other formula constraints that can be found within SMLP's workflow are:

- theta (θ) : Introduces a constraint that defines a circular region around a solution that is being studied. The radius that describes the circular region around a specific solutions (knob) p value variable is defined inside the specification file and it is used within the formula as: The expression $\theta(p,p') = \|p - p'\| \leq r$, where p' is the solutions p value.
- objective : Multiple objectives can be defined in the specifications file. During runtime, the goal is to maximise the objective's lower value.

The specification file defines the free inputs, knobs and outputs of the system by specifying their value ranges and other details. The following are the specifications of various variables used in the system:

- `"label":"y1", "interface":"output", "type":"real"`
- `"label":"x1", "interface":"input", "type":"int", "range":[0,10]`
- `"label":"p1", "interface":"knob", "type":"real", "range":[0,10], "rad-rel":0.1, "grid":[2,4,7]`

Below are some examples of the alpha, beta and eta constraints that could be used inside the SMLP specifications file:

- `"alpha": "p2 < 5 and x1 == 10 and x2 < 12"`
- `"beta": "y1 >= 0 and y2 >= 0"`
- `"eta": "p1 == 4 or (p1 == 8 and p2 > 3)"`

1.1.2 Translating the neural network's layers to Z3 terms

Currently, SMLP supports neural networks, decision trees, random forests and polynomial models. When a neural network acts as an input to SMLP, it is crucial to translate its structure and parameters into a form that can be understood by the Z3 solver. A deep neural network typically consists of an input layer, hidden layers and an output layer. The network's layers are split and handled separately during the representation process. For each layer, the weights and biases are extracted and are used to define the linear transformation applied to the input data. In this case, we can consider a full connected dense layer, where the output for this layer can be described as:

$$y = W \cdot x + b$$

Where W is the matrix of weights, x is the input vector, b is the bias vector and y is the output vector of the layer.

The only activation functions that are currently handled in SMLP are the ReLU and the identity activation function in a dense layer. The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

In Z3 terms, this is represented using the ITE (if-then-else) expression:

$$\text{ReLU}(x) = \text{ITE}(x > 0, x, 0)$$

This expression ensures that if x is greater than 0, the output is x, otherwise, the output is 0.

When expressing y as a sum of $x_i \cdot w_i$:

$$y = \sum_i x_i \cdot w_i + b$$

When replacing these equations we get:

$$\text{ReLU}\left(\sum_i x_i \cdot w_i + b\right) = \text{ITE}\left(\sum_i x_i \cdot w_i + b > 0, \sum_i x_i \cdot w_i + b, 0\right) \tag{1.1}$$

1.1.3 Feature Scaling

If the input features were scaled before the training process, their corresponding terms within the final neural network Z3 representation are replaced with the unscaled version. MinMax scaling is used to normalise the data and to create a more robust training process. When applying the unscaling, each variable x_i is replaced by:

$$x_{i,\text{unscaled}} = x_{\min,i} + x_i \cdot (x_{\max,i} - x_{\min,i})$$

Where $x_{i,\text{unscaled}}$ is the unscaled input feature, x_i is the scaled input feature, $x_{\min,i}$ is the minimum value of the original feature x_i and $x_{\max,i}$ is the maximum value of the original feature x_i.

When applying the unscaling, each variable x_i is replaced by $x_{i,\text{unscaled}} = x_{\min,i} + x_i \cdot (x_{\max,i} - x_{\min,i})$.

Thus, equation 1.1 becomes:

$$\begin{aligned} y &= \text{ReLU}\left(\sum_i \left(x_{\min,i} + x_i \cdot (x_{\max,i} - x_{\min,i})\right) \cdot w_i + b\right) \\ &= \text{ITE}\left(\sum_i \left(x_{\min,i} + x_i \cdot (x_{\max,i} - x_{\min,i})\right) \cdot w_i + b > 0,\right. \\ &\qquad \left.\sum_i \left(x_{\min,i} + x_i \cdot (x_{\max,i} - x_{\min,i})\right) \cdot w_i + b, 0\right) \end{aligned} \tag{1.2}$$

The same process is followed when expressing the final layer (i.e. output layer) using Equation 1.2 to create the output variables $\{y_i\}$.

1.1.4 Using NN-Z3 and constraints

To describe the final layer of the neural network using the weights of the previous layers, we express each output variable y_i in terms of the ReLU activations and weighted sums of the inputs from the previous layers. For instance, the outputs of the final layer can be written as:

$$y_0 = \text{ReLU}\left(\sum_i w_{0i} \cdot x_i + b_0\right), \quad y_1 = \text{ReLU}\left(\sum_i w_{1i} \cdot x_i + b_1\right), \quad \text{and so on.}$$

These expressions are then translated into Z3 equations to create a representation suitable for the solver. These equations are added to the Z3 solver to model the behavior of the neural network accurately. Additionally, extra constraints can be imposed on the Z3 solver to enforce specific conditions or requirements. These constraints may include bounds on the input variables, conditions on the outputs, or other logical assertions necessary for the particular problem being solved. By incorporating these constraints, we ensure that the Z3 solver provides solutions that adhere to the desired properties and constraints of the neural network model.

1.2 PySMT formulas

Integrating PySMT into the main SMLP workflow requires addressing several complexities due to the specific demands of this tool. PySMT within the realm of its SMLP usage, acts as the mediator by formulating complex constraints that are going to be imposed onto the neural network verifier, which in our case is going to be Marabou. Marabou's Python API is very restrictive in terms of how these constraints can be described and passed on to the verification stage. To address these challenges, tailored algorithms and procedures were developed, ensuring that PySMT and Marabou work seamlessly together within the SMLP pipeline.

The aforementioned methodologies cover the handling of different types of constraints, managing the flow of information between the tools, and ensuring that the overall system remains efficient and scalable (in the future newer neural network verifiers different to Marabou may be integrated). The successful integration of these tools within SMLP not only enhances the verification capabilities of the platform but also opens up new possibilities for applying formal methods to complex machine learning models (like neural networks with activation functions other than ReLU).

This section discusses the algorithms and methodologies that were designed in order to seamlessly integrate both PySMT, Marabou and connecting these two tools to successfully allow verification operations to be executed through this new pipeline.

1.2.1 Text Parser

Most of the formulas that will be applied and used in the verification step are instantiated through the specification file. This file includes the formulas in a string format. Thus, an appropriate conversion has to be made in order to be able to edit and process

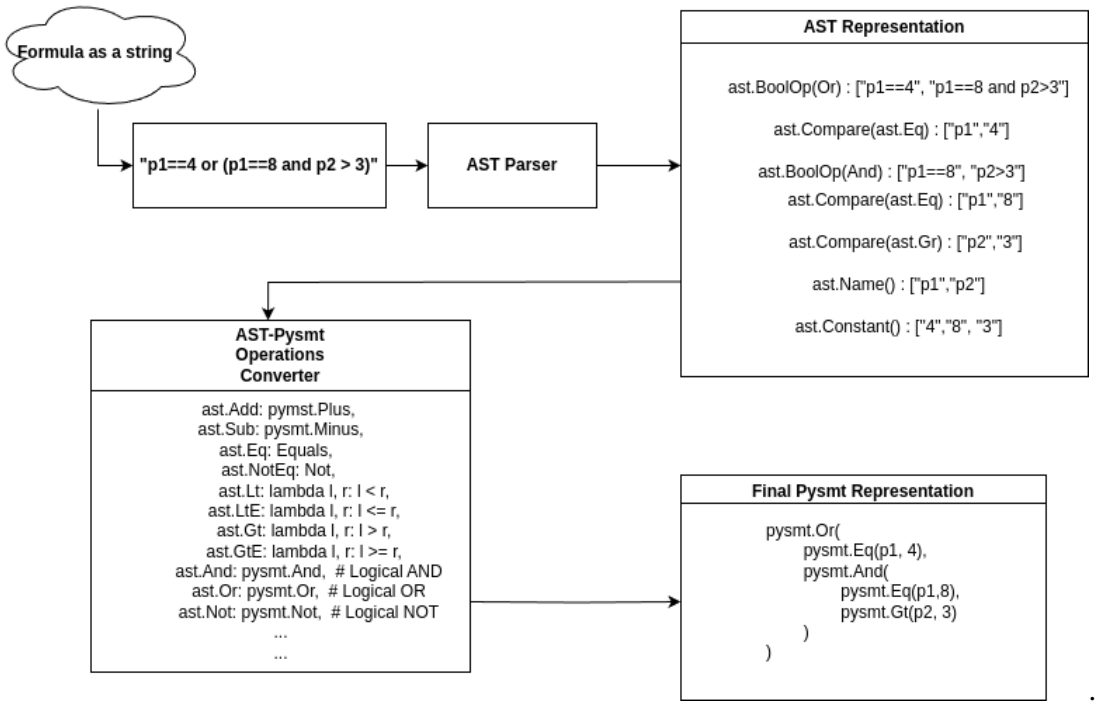

Figure 1.2: Converting strings to PySMT formulas

the aforementioned formulas. A parser has been created for this purpose, parsing each formula independently during runtime and converting it into its equivalent PySMT formula, allowing further access to its properties.

The initial step of converting the formulas into a string format is using Python's Abstract Syntax Trees (AST) library, which will convert the string into a more meaningful representation. After this step has been completed, AST will have successfully introduced an Object Oriented approach to the task, assigning all components of the formula a more meaningful representation.

Hence, an intermediary translation block is created (as seen in Fig. 1.2) to convert the now meaningful AST formulas into its equivalent PySMT formula. This new representation allows the creation of a one-to-one translation to the equivalent classes that PySMT offers. A recursive operation processes the components of the formula from top to bottom and replaces each AST structure with its equivalent PySMT representation, creating a final concrete structure.

1.2.2 VNN-Lib Parser

As mentioned in Chapter **??**, VNN-Lib is widely adopted by most state-of-the-art neural network verifiers as the standard format for describing neural network verification specifications (input/output variables, variable bounds, constraints represented in formulas etc.) . Given that most formulas in the workflow are now represented as PySMT formulas, they can be processed and translated into other equivalent formats, including VNN-Lib.

By breaking down the formula into sub-formulas and further traversing deep into the formula's structure, we manage to extract vital nodes that are going to be directly translated into their VNN-Lib equivalent representation. The recursive nature of the algorithm ensures that the nested structure of the formulas is preserved during this conversion.

For example, a formula represented as a PySMT formula and as an `AND` node with two children $(x \geq 1)$ and $(y < 5)$ would be converted back into the VNN-Lib format as:

$$(\text{and } (\geq \text{ x } 1) \ (< \text{ y } 5))$$

This capability to convert between PySMT's internal representation and VNN-Lib seamlessly ensures that the formulas can be easily exchanged between different tools and frameworks, preserving their logical structure and meaning. The VNN-Lib parser is a crucial component of our workflow, enabling the integration of VNN-Lib with PySMT and other neural network verification tools that may be added in the future.

1.2.3 Handling Disjunctions-Conjunctions

When initiating the verification phase, we must first gather all the formulas that are going to be imposed onto the neural network's constraint equation list. These formulas may usually have a complex formulation containing both disjunctions and conjunctions. A logical splitting of each formula into distinctive components is essential in order to iteratively process and apply them to the neural network verifier.

Disjunctions allow the representation of the logical 'OR' conditions, where at least one of the sub-formulas must be true. The first step in the pre-processing procedure is the extraction of disjunctions at the top level of the formula. Disjunctions in the formulas are processed and transformed so that they are represented as linear inequalities, which will bring the equation into a format where each variable is accompanied by

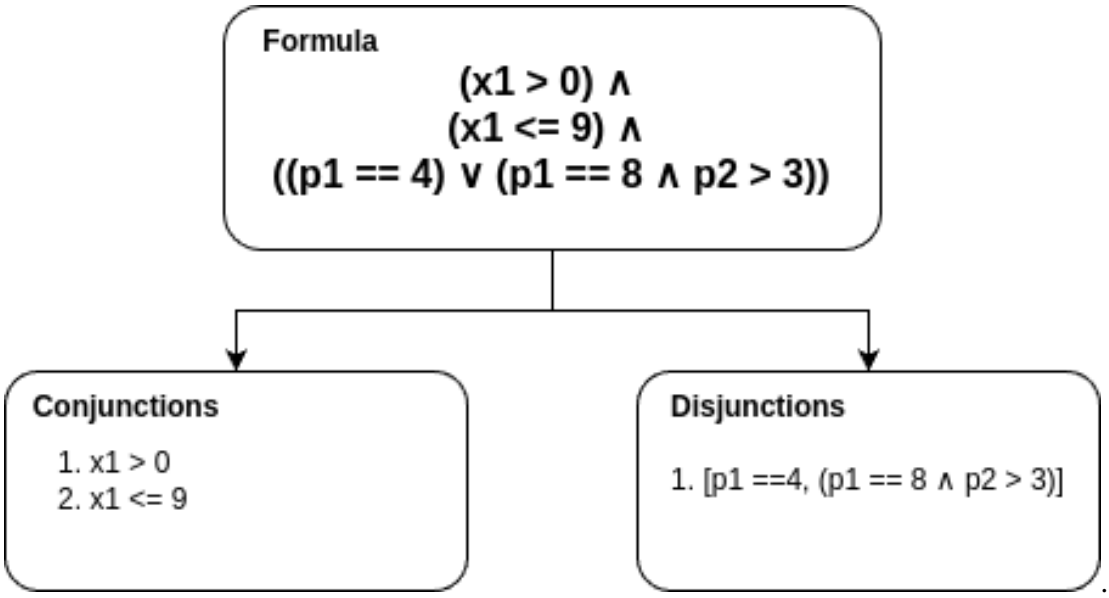

Figure 1.3: Splitting PySMT formula into conjunctions and disjunctions

a coefficient and is linearly connected to other variables with subtraction or addition operations. Examples of where disjunctions are used are mentioned in Chapter 1.3.4. Certain features in the specification file (Chapter 1.1.1), such as the grid, specifying a range of permissible values for an input parameter, can be represented through a disjunction in which all values are represented as a linear constraint sub-formula. These disjunctions, after the pre-processing stage are then connected to the Marabou Disjunction API at which point they are appended into the system exploration parameters.

Conjunctions, on the other hand, represent the logical ‘AND‘ conditions and are also handled and applied separately. Each conjunctive component of the formula must be true for the parent formula to hold. In practice, this means that each linear inequality (or equality) part of a more prominent conjunction, is processed individually and appended to the neural network verifier’s system exploration constraints. This allows the verifier to satisfy all constraints and conditions concurrently during execution.

Fig 1.3 shows an example of how a PySMT formula is logically split into a list of conjunctions and disjunctions. This step is necessary as we must process each subexpression independently when performing the Marabou translation process in the later stages.

1.2.4 Z3 Simplifier

One of the main motivations behind choosing the PySMT library as the interface to build the formulas is its built-in functionality of expressing them in SMT-LIB. This functionality allows the formulas to be directly plugged into state-of-the-art SMT solvers during runtime. Furthermore, state-of-the-art solvers like Z3 offer additional features like formula simplifiers, which reformat and restructure formulas, converting them to a more meaningful and simple representation. Alongside the simplification techniques being applied to the formulas, there is also the option to further reformat them into several supported forms. The supported formats include Conjunctive Normal Form (CNF), Negation Normal Form (NNF), and Disjunctive Normal Form (DNF), each of which has specific advantages depending on the type of problem being solved.

For our purposes, the Z3 simplifier is used in conjunction with PySMT's integrated functionalities and plays a vital role in restructuring complex formulas into a format digestible by Marabou's API interface. The initial step of parsing and reformatting formulas before being applied as a constraint on the neural network verifier is described in Chapter 1.2.3, where we apply an algorithm that logically separates any given formula at the highest level into distinct conjunctive and disjunctive formulas.

The second phase of this pre-processing algorithm focuses on further optimising the formula structure by eliminating unnecessary calculations. This includes simplifying long chains of floating-point operations and condensing repetitive or redundant expressions. For example, a formula like $x + x \leq 2$ can be simplified to $2x \leq 2$, and further reduced to $x \leq 1$. By minimising the complexity of formulas in this manner, we not only reduce the potential for numerical errors but also eliminate the risk of mistakes that could arise if a custom approach were set in place to simplify the PySMT formulas.

The simplified formulas, now in Z3 format, are then converted back into their equivalent SMT-LIB representation. We then leverage PySMT's functionality to parse the SMT-LIB code into PySMT formulas without any loss of information. More concrete examples of the Z3 simplifier in use are described in the next section.

1.2.5 Objectives Function (ITE & ITE negation)

Whilst Chapter 1.2.4 covers the majority of use cases, there are more complex scenarios which require further processing of the formulas and reformatting them into an equivalent representation. Due to the nature of the formulas and the difficulty of

directly expressing pysmt expressions to marabou constraints caused by the API's limitations, a customised approach for each use case must be developed in order to bring the expressions in a Marabou-encoded format.

Namely, one of the PySMT features that require custom re-formatting and handling is the if-then-else case. In this scenario a formula that utilises the if-then-else (ITE) operation will appear as:

$$\text{ITE(condition, true\_expression, false\_expression)} \geq c \Rightarrow$$
$$\text{(condition ? true\_expression : false\_expression)} \geq c.$$

Currently, Marabou's API does not directly support the assertion of formulas that contain `ITE` sub-expressions, and thus re-formatting is required. We can rewrite the formula in this format:

$$(\text{condition} \wedge \text{true\_expression} \geq c) \vee (\neg\text{condition} \wedge \text{false\_expression} \geq c)$$

This allows us to treat the ITE expressions as disjunctions, which are directly supported through Marabou's API. However, all 3 expressions (condition, true_expression, false_expression) may introduce complexity into the encoding process as they may not be in a linear inequality format as described in Chapter 1.2.3. For this purpose, we will utilise the power of Z3's simplifier functionality and encode the initial state of the ITE formula into its equivalent SMT-Lib representation and query for a simplified version to be generated.

However, `ITE`s are often part of a larger formula and function as sub-expressions within it. An example of an `ITE` being used is:

$$(y1 \geq 0) \wedge (y2 \leq 8) \wedge (\texttt{ITE}(\text{condition}, \text{true\_expression}, \text{false\_expression}) \geq 0)$$

It is also a common use case that we will require the negation of this formula while searching for a counterexample in the verification process. The negation of such a query introduces increased complexity because the result of negating this formula is:

$$(y1 < 0) \vee (y2 > 8) \vee (\texttt{ITE}(\text{condition}, \text{true\_expression}, \text{false\_expression}) < 0)$$

Expanding the `ITE` expression, the formula becomes:

$$(y1 < 0) \vee (y2 > 8) \vee ((\text{condition} \wedge \text{true\_expression} < 0) \vee (\neg\text{condition} \wedge \text{false\_expression} < 0))$$

While this formula is logically sound, it introduces the issue of inserting a conjunction into the parent-level disjunction, which is not supported by Marabou's API. This necessitates the elimination of the nested conjunction and rewriting the formula into Disjunctive Normal Form (DNF).

We can describe the current state of the formula as:

$$\text{Or}(A,B,C) \quad \text{where} \quad C = \text{And}(\text{Or}(D,E),\text{Or}(F,G)) \tag{1.3}$$

Let $K = \text{Or}(D,E)$, then $C = \text{And}(K,\text{Or}(F,G))$, which is equivalent to:

$$\text{Or}(\text{And}(K,F),\text{And}(K,G)) \tag{1.4}$$

Using 1.4:

$$\text{And}(K,F) = \text{And}(F,\text{Or}(D,E)), \quad \text{which is equivalent to :}$$

$$\text{Or}(\text{And}(F,D),\text{And}(F,E)) \tag{1.5}$$

Applying the same logic to:

$$\text{And}(G,K) = \text{Or}(\text{And}(G,D),\text{And}(G,E)) \tag{1.6}$$

Finally, combining equations 1.3, 1.5, 1.6 together:

$$\text{Or}(\text{And}(K,F),\text{And}(K,G)) = \text{Or}(\text{Or}(\text{And}(F,D),\text{And}(F,E)),\text{Or}(\text{And}(G,D),\text{And}(G,E))),$$

which simplifies to:

$$\text{Or}(\text{And}(F,D),\text{And}(F,E),\text{And}(G,D),\text{And}(G,E)) \tag{1.7}$$

Consequently, our initial equation now becomes:

$$\text{Or}(\text{A},\text{B},\text{Or}(\text{And}(F,D),\text{And}(F,E),\text{And}(G,D),\text{And}(G,E))) \tag{1.8}$$

Which can be simplified to:

$$\mathrm{Or}(\mathrm{A}, \mathrm{B}, \mathrm{And}(F,D), \mathrm{And}(F,E), \mathrm{And}(G,D), \mathrm{And}(G,E)) \tag{1.9}$$

Using 1.9, we can now successfully apply the objective function constraint onto Marabou through the Disjunction API.

1.3 Marabou

One of the key issues in this integration is the need to bridge the gap between PySMT's flexibility in creating formulas that are ultimately going to be used as Marabou's constraints, and Marabou's rigid API requirements. The procedures that have been developed for this cause involve breaking down PySMT's complex constraints into simpler and more manageable formulas that can be efficiently represented as Marabou constraints. Additionally, these procedures ensure that the constraints that are being passed through the SMLP workflow without loss of information by creating formulas are reformatted in a new formula, which is critical in maintaining the integrity of the neural network verification process.

Marabou will now serve as the primary solver for neural network verification tasks, processing formulas represented in the PySMT format. However, due to the complexity of the system, specific steps must be taken to address issues that may arise from the mechanisms and approaches used by SMLP prior to the verification process. These issues include the use of fractions when constructing formulas, scaling variables during the training phase, the requirement that certain variables be treated as integers and predefining as set of values that a variable is permitted to take on.

1.3.1 Converting fractions to floats

In the current SMLP's iteration where Z3 is used in the context of formal verification, the system leverages Z3's precision as it operates in exact arithmetic with the use of fractions. Fractions allow the system to maintain high accuracy in calculations as they are lossless when solving constraints. This level of precision is crucial when exact solutions are required, particularly in systems where even minor inaccuracies could lead to significant errors. However, Marabou uses floating-point arithmetic, which is generally faster and more efficient for large scale computation, but introduces a level of approximation that inevitably leads to precision loss. In the context of neural network

verification where the neural network's weights and biases are represented as floating point numbers, it is not expected to have disastrous effects.

To accommodate for Marabou's restrictions in arithmetic representation, it is necessary to convert the precise fractional representations that would be used by Z3 into floating-point numbers that Marabou can recognise. As mentioned earlier, this process introduces a precision loss which is caused by the fact that floating-point arithmetic cannot represent all fractions exactly. This loss of information creates challenges whilst replacing Marabou as the main neural network verifier.

An example that highlights this challenge of ensuring accuracy when translating exact arithmetic fractions to floating-point numbers is the fraction $\frac{1}{3}$. Whilst Z3 handles this with exact precision, the conversion cannot be exactly represented by a floating-point number. Typically, the approximation in this instance would be 0.33333333, which introduces a small but significant difference from the original value of 0.33333333... initially.

1.3.2 Scaled/Unscaled Variables

As previously discussed in Chapter 1.1.3, it is usually the case that before training the neural network (Chapter **??**) that is going to be used in the neural network verification step, we scaled the training data using the MinMax algorithm. It is important to note that all constraints and assertions mentioned in the specifications file and all the formulas that are formed within the workflow, concern the unscaled variables of the system. Thus, when instantiating the Marabou interface class, it is essential to keep track of both categorisations of variables as PySMT distinguishes between the two by specifying whether the variables in question are scaled or unscaled.

Upon instantiating the Marabou class and loading the neural network into the solver, a list of free variables is automatically generated. These variables consist of the input, ReLU and output variables. Each variable is referenced through the index which is located within the neural network variable list. The input and output variables reference the scaled version as Marabou is unaware of the scaling performed to the network beforehand. This raises the need to introduce the new unscaled variable into Marabou through its interface and correctly map the pairs of scaled/unscaled variables. The new unscaled auxiliary variables are assigned their own indices and will later be referenced by them.

Upon the insertion of a new constraint into the system and during the translation

phase where PySMT formulas become Marabou constraints, the variables that are referenced within the formula must be translated to the newly inserted unscaled equivalent variables. Using the correct index for each unscaled variable, the new assertions can be added into Marabou's constraint list.

In Table 1.1, we can see how Marabou encodes the neural networks variables internally and how they can be accessed by referring to their corresponding index in the encoding table. We can also notice that in the end of the table, Marabou has appended the newly added unscaled auxiliary variables upon which we will be applying the constraints to.

To create the correlation between each pair of scaled and unscaled variables, and for Marabou to propagate the constraints on the unscaled variable to the scaled version during the satisfiability step, equations need to be created that describe their relationship. Using the equation described in Chapter 1.1.3, we can successfully apply constraints to the unscaled auxiliary variables without directly referencing the scaled variables (x0,x1,p0,p1,y0,y1).

x0	x1	p0	p1	ReLu Auxiliaries	y0	y1	$x0_{\text{unsc}}$	$x1_{\text{unsc}}$	$p0_{\text{unsc}}$	$p1_{\text{unsc}}$	$y0_{\text{unsc}}$	$y1_{\text{unsc}}$
0	1	2	3	...	28	29	30	31	32	33	34	35

Table 1.1: Marabou constraints/neural network variables encoding table

1.3.3 Adding fixed bounds on a variable

A very common use case when applying constraints on a specific variable is setting its lower and upper bounds. These bounds can be derived from the PySMT formula, as they are distinct components of the formula and are processed independently. These bounds have 5 different formats in which they can appear within the formula: $>, <$ $, <=, >=, =$. Each of these operations defines a constraint on a variable, ultimately affecting either the lower bound, upper bound or both.

When dealing with $'>='$ *or* $'<='$ operators (weak inequalities), the process involves setting the lower or upper bound to the floating-point number c which is specified in the comparison operation with the variable. However, when dealing with $'>'$ or $'<'$ operators (strict inequalities), we must consider that since we are not dealing with exact arithmetic which allow for exact representations, floating-point number must represent this operation. Inferentially, a floating-point number c must be nominated as a lower or upper bound.

An example that replicates this issue is the strict inequality $x > 2$, where it is necessary to find a floating-point number c to be set as a lower bound to variable x. The number 2 can be represented as a floating-point number, but to apply a strict inequality like $x > 2$ we must find the next smallest floating-point number after 2. This can be achieved through various methods, which can provide the closest representable number greater than 2, ensuring this way that the strict inequality is correctly maintained in floating-point operations. In this example, the number c which is going to be set as a lower bound for variable x is 2.0000000000000004, as it is the next smallest floating-point number after 2.

Another very common use case is the use of strict equality, where the constraints specify that a given variable must take on the specified value. This is made possible through Marabou's API in which you can define linear equations of this format:

$$\sum_i a_i x_i = c \tag{1.10}$$

where a0 is 1, x0 is the variable referenced in the formula and c is the specified value.

All aforementioned comparison operators (inequalities, equalities) are described as constraints in Marabou via its API and are incorporated into the network's list of equations, guiding the solving process.

1.3.4 Handling Integers

In Chapter 1.1.1 the possibility of certain variables entering the system with different types (real, integer) is discussed. When dealing with integers its important to note that Marabou does not directly support the use of neural network inputs and outputs as integers, thus a workaround must be used in this case.

The solution to this issue is the use of disjunctions in which each component describes a discrete value that the variable can take on. A unique disjunction can be created for each integer variable within the system that expresses the range of applicable integer values. Disjunctions are directly supported in Marabou's API and are discussed more deeply in Chapter 1.2.3. An example of how disjunctions can be used in this case:

$$x = 1 \vee x = 2 \vee x = 3 \vee x = 4 \tag{1.11}$$

However, whilst this approach effectively circumvents Marabou's limitations, it also comes with a caveat as it introduces vulnerabilities in the potential scalability of

the system. This is due to the fact that certain variables may take on a large range of integer values, leading to increased complexity and longer evaluation times during execution because the disjunction formula proportionally grows in size. To date, no alternative solutions have addressed this issue comprehensively, making this a notable limitation to our approach.

1.3.5 Marabou solving

After all the aforementioned steps and use cases have been correctly handled and constraints applied onto the Marabou solver, the last phase of the verification process is to query the solver and request the initiation of the solving process. Once the verification problem has been finalised and correctly defined, Marabou will use its internal processes and solver to explore the state space of the neural network.

The solver systematically checks whether the specified constraints, which involves defined bounds on all input and output variables, and well-defined disjunctions, can be simultaneously satisfied. In the event that Marabou finds a solution (witness), this implies that there exists an input configuration for which the neural network's outputs satisfy the given constraints. Conversely, if no witness is found, the solver will conclude that the network cannot satisfy the constraints under any valid input.

Marabou's response to the `solve` query, is typically binary:

- SAT (Satisfiable): This indicates that Marabou has found an input configuration (a witness) that satisfies all the constraints. In other words, the neural network behaves as expected under at least one input scenario that meets the defined properties.
- UNSAT (Unsatisfiable): This indicates that no such input configuration exists, meaning the network cannot satisfy the constraints under any circumstances.

In any other event where something can go wrong in the witness search process, the solver will raise an error instead of responding with SAT or UNSAT, and this raises the need for a customised handling approach that explores the different reasons that caused the solver to fail in responding.

1.4 PySMT and Marabou integration

In this section we will be describing the architecture of the mechanisms that have been placed inside the SMLP workflow and how they work together to navigate the

verification process. The properties and responsibilities encompass several key tasks, such as:

- Runtime Decision-Making: coordinating and dynamically deciding which settings and representations to apply during the verification process. This includes deciding whether to use Z3 terms or PySMT formula representation based on the context and the specific requirements that are represented in the form of system configurations.

- Formula Forwarding and Solver Selection: Once the appropriate settings have been set, the mechanism forwards the selected formulas to the appropriate solvers. Currently, due to the work that has been introduced in the previous chapters, the system can now support the Marabou neural network verifier alongside with the pre-existing Z3 solver. This requires a deep integration between the formula representation interfaces and the solvers API interface, to ensure that the formulas are correctly interpreted and processed by the allocated solver.

- Customizable Verification Workflows: Each step in the verification process may differ depending on the solver being used. For example, while Z3 might require a certain handling of logical formulas, Marabou may need a different approach due to its specific capabilities and limitations. SMLP must be capable of handling customised approaches for each setting for each step withing the verification process.

1.4.1 Formula-Solver Interfaces

The integration of Marabou into the system relies on two essential components: the interface that constructs and re-formats PySMT formulas, and the PySMT-to-Marabou constraints parser. These components are critical for ensuring that the system supports the necessary features and are tightly coupled in order to maintain a consistent workflow without disrupting the core operations.

The PySMT formulas interface is responsible for:

- constructing the formulas.

- parsing formulas represented as strings with a customised text-to-PySMT parser (Chapter 1.2.1).

- eliminating negation operations by iteratively propagating them to the lower levels of the formula's structure.

- converting ITEs to a more manageable format (Chapter 1.2.5).

- integrating Z3's simplifier and utilising it in more complex scenarios (Chapter 1.2.4).

- extracting conjunctions and disjunctions from formulas during the last step before processing them as Marabou constraints (Chapter 1.2.3).

On the other hand, the Marabou interface is responsible for:

- creating and managing neural network variables and mapping them to the corresponding PySMT formula variables.

- generating auxiliary variables and creating equations to form pairs of scaled and unscaled inputs and outputs (Chapter 1.3.2).

- creating disjunctions to facilitate for the need of supporting integer neural network inputs/outputs (Chapter 1.3.4).

- replacing fractions with floating point numbers (Chapter 1.3.1).

- converting PySMT formulas into Marabou constraints.

- extract coefficients from variables in PySMT linear inequalities so that they can be utilised when describing Marabou constraints (Chapter 1.3.3).

- executing the solving process and processing the results into a readable format (Chapter 1.3.5).

Both of these interfaces work together to produce the final result of deciding on a witness that follows the constraints set by the system. Whilst the Marabou interface heavily relies on the fact that the formulas that are going to be processed into Marabou constraints are in PySMT format, few configurations are required to be applied to the current version in order to support other formula formats.

1.4.2 Integrating into the new workflow

Whilst both interfaces produce some level of abstraction in their usage, an additional interface is needed to decide which version of the system settings to trigger. The need for such interface stems from the fact that the two approaches, Z3 and Marabou require a different procedure during runtime.

This interface is responsible for redirecting the calls to the predefined functionalities set by each solver's suite of mechanisms, which are in place to allow further scalability and loosely coupled approach to the workflow's requirements. It is created following the singleton design pattern, allowing all instances to ascertain consistency in the data that is used throughout the complex workflow.

Its functionality as a "switch" class is integral to SMLP's verification pipeline as it is the first point of contact before directing the data flow towards the correct implementation. Each implementation contains the logic and tailored procedural actions for each individual process within the pipeline.

Bibliography

[BKK24] Franz Brauße, Zurab Khasidashvili, and Konstantin Korovin. Smlp: Symbolic machine learning prover. *arXiv preprint arXiv:2402.01415*, 2024.