UNIVERSITY OF GENOVA

PHD PROGRAM IN COMPUTER SCIENCE AND SYSTEMS ENGINEERING

Experimenting with Constraint Programming Techniques in Artificial Intelligence: Automated System Design and Verification of Neural Networks

by

Stefano Demarchi

Thesis submitted for the degree of *Doctor of Philosophy* (35° cycle)

May 2023

Armando Tacchella	Supervisor
Giorgio Delzanno	Head of the PhD program

Thesis Jury:

Dritan Nace, *Universitè de Compiègne*	External examiner
Roberto Sebastiani, *Università di Trento*	External examiner
Grażyna Suchacka, *Opole University*	External examiner

Dibris

Department of Informatics, Bioengineering, Robotics and Systems Engineering

Acknowledgements

The work presented in this Thesis would have never been possible without the support and the teaching of my supervisor Armando Tacchella, I am forever grateful for his utmost dedication and infinite patience. I also wish to thank all my colleagues in the lab: Marco Menapace introduced me to the world of elevators and helped me learn a ton about software development in practice, along with Giuseppe Cicala; Dario Guidotti on the other hand helped me understand the new context of neural networks and his work clearly and patiently. Frosina, Francesco, Massimo: our exchanges, fruitful and trivial, relieved the stress and fatigue. A special thanks to Luca Pulina and Danilo Calzetta, who supported me while preparing the admission.

I extend my gratitude to all the researchers I met during my last year, when it was finally possible to travel again after the COVID-19 pandemic: Prof. Dritan Nace who welcomed me at Universitè de Technologie de Compiègne, Guillaume Joubert who proposed new ideas, Joelle Al Hage with the valuable discussions we held and Stephane, Philippe, Maxime, Remy, Antoine, Thibault, Lyes, Sana, Ling, Soundouss and everyone I shared coffee and ideas with.

Last, but not least, I thank my family and friends who were always there during this journey, supporting and helping me. Erica, my beloved, who endured as much as me the weight of this work and still is crazy enough to try and start to be a researcher too. The whole San Giorgio Fight Team, a family before a sport team. All of you are very near and dear to me.

Abstract

This thesis focuses on the application of Constraint Satisfaction and Optimization techniques in two Artificial Intelligence (AI) domains: automated design of elevator systems and verification of Neural Networks (NNs). The three main areas of interest for my work are (i) the languages for defining the constraints for the systems, (ii) the algorithms and encodings that enable solving the problems considered and (iii) the tools that implement such algorithms.

Given the expressivity of the domain description languages and the availability of effective tools, several problems in diverse application fields have been solved successfully using constraint satisfaction techniques. The two case studies herewith presented are no exception, even if they entail different challenges in the adoption of such techniques. Automated design of elevator systems not only requires encoding of feasibility (hard) constraints, but should also take into account design preferences, which can be expressed in terms of cost functions whose optimal or near-optimal value characterizes "good" design choices versus "poor" ones. Verification of NNs (and other machine-learned implements) requires solving large-scale constraint problems which may become the main bottlenecks in the overall verification procedure.

This thesis proposes some ideas for tackling such challenges, including encoding techniques for automated design problems and new algorithms for handling the optimization problems arising from verification of NNs. The proposed algorithms and techniques are evaluated experimentally by developing tools that are made available to the research community for further evaluation and improvement.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Context and motivation

1.1.1 Automated system design

The problem of automating product configuration and design has a long history in diverse application fields. In late 1970s, possibly the first configuration program was developed for building computer systems meeting custom requirements by Digital Equipment Corporation McDermott (1981). Noticeably, among the earliest attempts to provide automated product configuration we find also elevators as a case study Marcus et al. (1987). But, to the best of our knowledge, the design of such systems has not been considered any further until recent works Annunziata et al. (2017); Demarchi et al. (2019). More in general, research in the configuration domain has flourished Zhang (2014) also thanks to the seminal contributions of Mittal, Frayman and Falkenhainer Mittal and Falkenhainer (1990); Mittal and Frayman (1989) who posed the foundations for the definition of configuration models. According to their definition, configuration is the task of combining different components with given *ports*, i.e., connections with other components, subject to an arbitrary number of constraints specifying the structure of the system and the compatibility between components at each port. The result is a set of components and the description of their connections.

In the same period, Franke (1998) suggests a set of input specifications that describe the configuration problem, together with the concept of *configuration objective*, i.e., a value that the configuration should meet. The idea of associating a value to a configuration in order to go beyond mere feasibility is also present in Brown (1998). Here, the author defines *design* as a complex process which includes configuration as a phase. However, design involves not only selection of components according to their compatibility, but also the the generation of values for their attributes, therefore refining the model and producing a more valuable result than a simple combination of parts. In this paper we view the configuration task as a *constraint satisfaction* problem, where one seeks any model compliant to the structural constraints, and the design task as a *constrained optimization* problem, i.e., the satisfaction problem plus cost functions to model the value of a configuration in terms of complete design.

On this subject, a tool named LIFTCREATE Annunziata et al. (2017) has been developed. This tool, starting from a database of commercial components, takes the designer from basic measurements, e.g., shaft size and payload, to a complete design guaranteeing feasibility within specific normative regulations — directive 2014/33/EU and related EN 81-20/81-50 norms. In particular, these norms introduce more stringent safety requirements for components in the elevator design with respect to their placement and accessibility requirements,

e.g., doors opening and car dimensions which are suitable for wheelchair access. The current problem-solving engine is based on special-purpose heuristics developed working with professionals in the domain of elevator design. This engine is coded to provide the user with a fast response while computing solutions with good perceived "quality", i.e., as close as possible to the ones that a human designer would conceive. In other words, while many feasible configurations may exists for the same design, the heuristic engine includes criteria to identify the best ones, but without giving strict optimality guarantees to avoid excessive computational burden. In the following, we quantify these criteria as *configuration objectives*, i.e., the value that the configuration should meet in order to be considered an acceptable design.

1.1.2 Neural networks verification

Adoption and successful application of deep neural networks (DNNs) in various domains have made them one of the most popular machine-learned models to date — see, e.g., Taigman et al. (2014) on image classification, Yu et al. (2012) on speech recognition, and LeCun et al. (2015) for the general principles and a catalog of success stories. Despite the impressive progress that the learning community has made with the adoption of DNNs, it is well known that their application in safety- or security-sensitive contexts is not yet hassle-free. From their well-known sensitivity to *adversarial perturbations* Goodfellow et al. (2015); Szegedy et al. (2014), i.e., minimal changes to correctly classified input data that cause a network to respond in unexpected and incorrect ways, to other less-investigated, but possibly significant properties — see, e.g., Leofante et al. (2018) for a catalog — the need for tools to analyze and possibly repair DNNs is strong.

As witnessed by an extensive survey Huang et al. (2018) of more than 200 recent papers, the response from the scientific community has been equally strong. As a result, many algorithms have been proposed for the verification of neural networks and tools implementing them have been made available. Some examples of well-known and fairly mature verification tools are Marabou Katz et al. (2019), a satisfiability modulo theories (SMT)-based tool that answers queries regarding the properties of a DNN by transforming the queries into constraint satisfiability problems; ERAN Singh et al. (2019), a robustness analyzer based on abstract interpretation and MIPVerify Tjeng et al. (2019), another robustness analyzer based on mixed integer programming (MIP). Other widely-known verification tools are Neurify Wang et al. (2018), a robustness analyzer based on symbolic interval analysis and linear relaxation, NNV Tran et al. (2020), a tool implementing different methods for reachability analysis,

Sherlock Dutta et al. (2019), an output range analysis tool and NSVerify Akintunde et al. (2018), also for reachability analysis. A number of verification methodologies — without a corresponding tool — is also available like Wu et al. (2018), a game-based methodology for evaluating pointwise robustness of neural networks in safety-critical applications. Most of the above-mentioned tools and methodologies work only for feedforward fully-connected neural networks with ReLU activation functions, with some of them featuring verification algorithms for convolutional neural networks with different kinds of activation function. To the best of our knowledge, current state-of-the-art tools are restricted to verification/analysis tasks, in some cases they are limited to specific network architectures and they might prove difficult to use for practitioners and, in general, those who are not familiar with the complete background.

Our tool NEVER2 finds itself at the intersection of the issues explained above, and aims to bridge the gap between learning and verification of DNNs. NEVER2 borrows its design philosophy from NEVER Pulina and Tacchella (2011), the first tool for automated learning, analysis and repair of neural networks. NEVER was designed to deal with multilayer perceptrons (MLPs) and its core was an abstraction-refinement mechanism described in Pulina and Tacchella (2010, 2012). As a system, one peculiar aspect of NEVER was that it included learning capabilities through the SHARK Igel et al. (2008) library. Concerning the verification part, NEVER could utilize any solver integrating Boolean reasoning and linear arithmetic constraint solving — HYSAT Franzle et al. (2007) at the time. A further peculiarity of the approach was that NEVER could leverage abstract counterexamples to (try to) repair the MLP, i.e., retrain it to eliminate the causes of misbehaviour. NEVER2 relies on the PYNEVER API Guidotti et al. (2021) and a first description of the system is available in Guidotti et al. (2020b), where the verification capabilities were provided by external tools like Marabou, ERAN and MIPVerify. This Thesis describes the new abstraction-refinement procedure implemented by NEVER2 and formalizes all the theorems involved. The version of NEVER2 corresponding to this work is available online Guidotti et al. (2022) under the Commons Clause (GNU GPL v3.0) license.

1.1.3 Research questions

Formally, this Thesis poses the following research questions:

Automated System Design

- (i) How can we choose among different alternatives to encode design subtasks into sets of constraints?
- (ii) How different constraint-based tools and their encoding impact the performance of the design subtasks?
- (iii) How can we support declarative encodings in an interactive web application?

Verification of Neural Networks

- (iv) How can we improve existing star-based verification techniques for neural networks?
- (v) What is missing from other verification tools, and how can we improve that?

All the work inspired from these questions and the existing background resulted in the following publications:

- D. Guidotti, **S. Demarchi**, *Counter-Example Guided Abstract Refinement for Verification of Neural Networks*, in Cyber-Physical Systems Summer School workshop, CPSWS 2022, Pula, Italy, September 19, 2022, Proceedings, 2022.
- **S. Demarchi**, D. Guidotti, A. Pitto and A. Tacchella, *Formal Verification of Neural Networks: a Case Study about Adaptive Cruise Control*, in International Conference on Modelling and Simulation, ECMS 2022, Aalesund, Norway, May 30th-June 3rd, 2022, Proceedings, 2022.
- G. Cicala, **S. Demarchi**, M. Menapace, L. Annunziata and A. Tacchella, *A Comparison of Declarative AI Techniques for Computer Automated Design of Elevator Systems*, in Intelligenza Artificiale 16 (1), 131-150, 2022
- **S. Demarchi**, M. Menapace and A. Tacchella, *Automated Design of Elevator Systems: Experimenting with Constraint-Based Approaches*, in International Conference of the Italian Association for Artificial Intelligence, AIxIA 2021, Online, Proceedings, 2022.

- **S. Demarchi**, M. Menapace and A. Tacchella, *Automating Elevator Design with Satisfiability Modulo Theories*, in IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, Oregon, November 4-6, 2019, Proceedings, 2019.

1.2 Contribution

The contribution of this Thesis is twofold. First, the results of the experiments on different ways of encoding design problems and how to employ constraint satisfaction techniques in verification of NNs contribute to deepen the knowledge in how different ways of encoding the constraints impact on the results. Second, for both case studies we developed special purpose tools that provide access to all the methodologies taken in exam. In particular, we contribute with the following results:

Declarative encodings for elevator systems. Starting from the work presented in Demarchi et al. (2019) we investigated how to benefit from declarative encodings in the elevator domain, and how to transfer such knowledge to generic configuration systems. In Demarchi et al. (2021) and Cicala et al. (2022) we experimented with several declarative encodings based on Genetic Algorithms (GAs), Satisfiability Modulo Theories (SMT) and Constraint Programming (CP) showing the strength and weaknesses of these approaches against the different aspects of the design task.

Neural networks verification in safety-critical domains. We use SMT as a pivot to deal with the verification of machine learning models with neural networks. We created a standard for the definition of benchmarks which uses SMT as the language for the definition of the verification properties Guidotti et al. (2023) and we contribute with two new case studies, i.e., Adaptive Cruise Control (ACC) Demarchi et al. (2022) and drone hovering. We also experiment with optimization algorithms for our verification procedure Demarchi and Guidotti (2022).

State of the art tool development. Alongside the encodings and the experiments herewith reported, we also developed new tools for providing fast and easy access to the algorithms. For the design of elevator systems we perfectioned the tool LIFTCREATE, which provides an interface towards the different encodings. Using this tool, the experiments shared the same overhead and provided reliable and replicable results.

In the verification topic, we created a software portfolio named *NeVerTools*[1] which groups the former contributions in the domain, packed in the PYNEVER Python API and two Graphical User Interfaces (GUIs), namely COCONET and NEVER2.

[1] https://github.com/NeVerTools

1.3 Overview

The manuscript is structured as follows. Chapter 2 resumes the relevant background notions for elevator systems, neural networks and the main techniques employed. In the first part, Chapter 3 details the modeling of elevator systems with variables, constraints and objectives and Chapter 4 enumerates the different encodings that we propose for replacing the current engine. After showcasing the web application for elevator design in Chapter 5, in Chapter 6 we show our experimental campaign for the search of a new declarative encoding.

In the second part we focus on neural networks, and we meticulously detail the abstraction algorithms in Chapter 7. Then we detail the tools we developed in order to foster research and collaboration in Chapter 8 and we collect the results of this topic in Chapter 9. Finally, Chapter 10 concludes the Thesis and sheds some light on possible future research perspectives.

Chapter 2

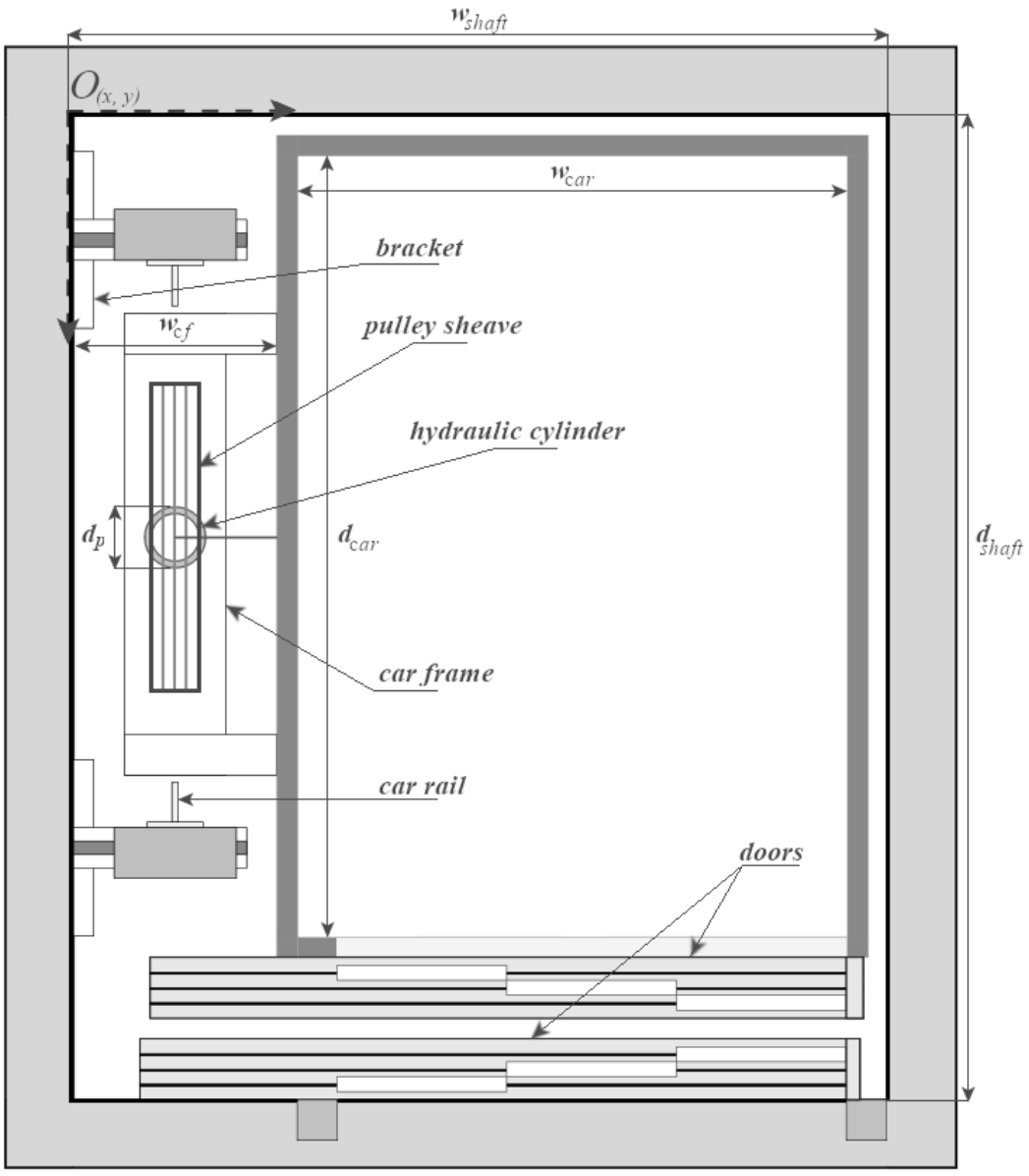
Background

2.1 A primer on elevator systems

Elevator systems are characterized by different lifting mechanisms, e.g., hydraulic cylinders or electric motors, and different setups, e.g., one or two cylinders, presence or absence of counterweights, and dedicated machine room versus machine-room-less implements. In this paper, we focus on Roped Hydraulic Elevators (RHEs), in which the lifting power is provided by hydraulics. In Fig. 2.1 we show the cross-section of a RHE similar to those produced by LIFTCREATE. Note that the image has been modified by removing some parts while enhancing others to ease understanding by a non-technical audience. The plan view accounts for the main components to be found in the design of an elevator. In the figure, the *shaft* denotes the enclosure space in which the elevator is installed. Inside the shaft, we can observe the *hydraulic cylinder*, providing lifting power, and the *car* attached to the *car frame*, a mechanical structure that supports the car and connects it to the piston through *ropes* (not visible in the plan view). The car frame slides within the shaft along the *car rails*. The ropes are guided through a *pulley* attached on top of the piston: one end is secured to the shaft base, the other to the car frame. Finally there are two kinds of doors, namely the *car door* and the *landing door*. As the name implies, the car door is only one and it is attached to the car, while there is one matching landing door for each floor.

Design of RHEs is usually performed in steps. The first one is the choice of the doors and the car frame, whose positioning must be determined considering shaft size, encumbrances, and tolerances since a minimum distance between moving and fixed parts must always be taken into account. In the second step, considering other parameters like the distance between the car and the shaft or the materials used for the car, an overall suspended weight and a maximum payload can be computed. Based on these findings, other components like ropes, rails, safety gear and the hydraulic cylinder can be engineered, making sure that compliance to the norms and regulations is always respected. In this phase, review of the previous phases might be necessary because some choices might not result in feasible solutions, which often makes the (manual) process to obtain the final design an iterative trial-and-error endeavor. Considering the plan view of the of RHEs in Fig.2.1, in this work we focus on elevators whose car frame is installed on the left side and doors are installed at the bottom of the drawing. The component selection we consider is limited to car frame, doors and hydraulic cylinder, whereas placement involves car frame and doors.

Figure 2.1 Cross-section (plan view) of a configured RHE. The shaft is the gray box surrounding the other components, the car frame is on the left side and doors at the bottom of the drawing.



2.2 Neural Networks

Basic notation and definitions. We denote n-dimensional *vectors* of real numbers $x \in \mathbb{R}^n$ — also *points* or *samples* — with lowercase letters like x, y, z. We write $x = (x_1, x_2, \ldots, x_n)$ to denote a vector with its *components* along the n coordinates. We denote $x \cdot y$ the *scalar product* of two vectors $x, y \in \mathbb{R}^n$ defined as $x \cdot y = \sum_{i=1}^{n} x_i y_i$. The *norm* $\|x\|$ of a vector is defined as $\|x\| = \sqrt{x \cdot x}$. We denote sets of vectors $X \subseteq \mathbb{R}^n$ with uppercase letters like X, Y, Z. A set of vectors X is *bounded* if there exists $r \in \mathbb{R}, r > 0$ such that $\forall x, y \in X$ we have $d(x,y) < r$ where d is the *Euclidean norm* $d(x,y) = \|x - y\|$. A set X is *open* if for every point $x \in X$ there exists a positive real number ε_x such that a point $y \in \mathbb{R}^n$ belongs to X as long as $d(x,y) < \varepsilon_x$. The complement of an open set is a *closed* set — intuitively, one that includes its boundary, whereas open sets do not; closed and bounded sets are *compact*. A set X is *convex* if for any two points $x, y \in X$ we have that also $z \in X \ \forall z = (1 - \lambda)x + \lambda y$ with $\lambda \in [0, 1]$, i.e., all the points falling on the line passing through x and y are also in X. Notice that the intersection of any family, either finite or infinite, of convex sets is convex, whereas the union, in general, is not. Given any non-empty set X, the smallest convex set $\mathscr{C}(X)$ containing X is the *convex hull of* X and it is defined as the intersection of all convex sets containing X. A *hyperplane* $H \subseteq \mathbb{R}^n$ can be defined as the set of points

$$H = \{x \in \mathbb{R}^n \mid a_1 x_1 + a_2 x_2 + \ldots + a_n x_n = b\}$$

where $a \in \mathbb{R}^n$, $b \in \mathbb{R}$ and at least one component of a is non-zero. Let $f(x) = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n - b$ be the affine form defining H. The *closed half-spaces associated with* H are defined as

$$H_+(f) = \{x \in X \mid f(x) \geq 0\} \qquad H_-(f) = \{x \in X \mid f(x) \leq 0\}$$

Notice that both $H_+(f)$ and $H_-(f)$ are convex. A *polyhedron* in $P \subseteq \mathbb{R}^n$ is a set of points defined as $P = \bigcap_{i=1}^{p} C_i$ where $p \in \mathbb{N}$ is a finite number of closed half-spaces C_i. A bounded polyhedron is a *polytope*: from the definition, it follows that polytopes are convex and compact in $\mathbb{R}^n$.

Neural networks. Given a finite number p of functions $f_1 : \mathbb{R}^n \to \mathbb{R}^{n_1}, \ldots, f_p : \mathbb{R}^{n_{p-1}} \to \mathbb{R}^m$ — also called *layers* — we define a *feed forward neural network* Abdi et al. (1999) as a function $\nu : \mathbb{R}^n \to \mathbb{R}^m$ obtained through the compositions of the layers, i.e., $\nu(x) = f_p(f_{p-1}(\ldots f_1(x)\ldots))$. The layer f_1 is called *input layer*, the layer f_p is called *output*

layer, and the remaining layers are called *hidden*. For $x \in \mathbb{R}^n$, we consider only two types of layers:

- $f(x) = Ax + b$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ is an *affine layer* implementing the linear mapping $f : \mathbb{R}^n \to \mathbb{R}^m$;
- $f(x) = (\sigma_1(x_1), \ldots, \sigma_n(x_n))$ is a *functional layer* $f : \mathbb{R}^n \to \mathbb{R}^n$ consisting of n *activation functions* — also called *neurons*; usually $\sigma_i = \sigma$ for all $i \in [1, n]$, i.e., the function σ is applied componentwise to the vector x.

We consider two kinds of activation functions $\sigma : \mathbb{R} \to \mathbb{R}$ that find widespread adoption: the *ReLU* function defined as $\sigma(r) = max(0, r)$, and the *logistic* function defined as $\sigma(r) = \frac{1}{1+e^{-r}}$. Although we do not consider them here, affine mappings can also represent convolutional layers with one or more filters Gehr et al. (2018). For a neural network $\nu : \mathbb{R}^n \to \mathbb{R}^m$, the task of *classification* is about assigning to every input vector $x \in \mathbb{R}^n$ one out of m labels: an input x is assigned to a class k when $\nu(x)_k > \nu(x)_j$ for all $j \in [1, m]$ and $j \neq k$; the task of *regression* is about approximating a functional mapping from $\mathbb{R}^n$ to $\mathbb{R}^m$. In this regard, neural networks consisting of affine layers coupled with either ReLUs or logistic layers offer universal approximation capabilities Hornik et al. (1989).

Verification task. Given a neural network $\nu : \mathbb{R}^n \to \mathbb{R}^m$ we wish to verify algorithmically that it complies to stated *post-conditions* on the output as long as it satisfies *pre-conditions* on the input. Without loss of generality[1], we assume that the input domain of ν is a bounded set $I \subset \mathbb{R}^n$. Therefore, the corresponding output domain is also a bounded set $O \subset \mathbb{R}^m$ because (i) affine transformations of bounded sets are still bounded sets, (ii) ReLU is a piecewise affine transformation of its input, (iii) the output of logistic functions is always bounded in the set $[0, 1]$, and the composition of bounded functions is still bounded. We require that the logic formulas defining pre- and post-conditions are interpretable as finite unions of bounded sets in the input and output domains. Formally, given p bounded sets $X_1, \ldots, X_p$ in I such that $\Pi = \bigcup_{i=1}^{p} X_i$ and s bounded sets $Y_1, \ldots, Y_s$ in O such that $\Sigma = \bigcup_{i=1}^{s} Y_i$, we wish to prove that

$$\forall x \in \Pi . \nu(x) \in \Sigma. \tag{2.1}$$

While this query cannot express some problems regarding neural networks, e.g., invertibility or equivalence Leofante et al. (2018), it captures the general problem of testing robustness

[1] Input domains must be bounded to enable implementation of neural networks on digital hardware; therefore, also data from physical processes, which are potentially ubounded, are normalized within small ranges in practical applications.

against *adversarial perturbations* Goodfellow et al. (2015). For example, given a network $\nu : I \to O$ with $I \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^m$ performing a classification task, we have that separate regions of the input are assigned to one out of m labels by ν. Let us assume that region $X_j \in I$ is classified in the j-th class by ν. We define an *adversarial region* as a set $\hat{X}_j$ such that for all $\hat{x} \in \hat{X}$ there exists at least one $x \in X$ such that $d(x,\hat{x}) \leq \delta$ for some positive constant δ. The network ν is *robust* with respect to $\hat{X}_j \subseteq I$ if, for all $\hat{x} \in \hat{X}_j$, it is still the case that $\nu(x)_j > \nu(x)_i$ for all $i \in [1,m]$ with $i \neq j$. This can be stated in the notation of condition (2.1) by letting $\Pi = \{\hat{X}_j\}$ and $\Sigma = \{Y_j\}$ with $Y_j = \{y \in O \mid y_j \geq y_i + \varepsilon, \forall i \in [1,n] \wedge i \neq j, \varepsilon > 0\}$. Analogously, in a regression task we may ask that points that are sufficently close to any input vector in a set $X \subseteq I$ are also sufficiently close to the corresponding output vectors. To do this, given the positive constants δ and ε, we let $\hat{X} = \{\hat{x} \in I \mid \exists x.(x \in X \wedge d(\hat{x},x) \leq \delta)\}$ and $\hat{Y} = \{\hat{y} \in O \mid \exists x.(x \in \hat{X} \wedge d(\hat{y},\nu(x)) \leq \varepsilon)\}$ to obtain $\Pi = \{\hat{X}\}$ and $\Sigma = \{\hat{Y}\}$. Notice that, given our definition, we consider adversarial regions and output images that may not be convex.

Reinforcement Learning. Reinforcement Learning (RL) is a machine learning approach whose aim is to teach agents how to solve specific tasks by trial and error, i.e., having them interact with an environment and then rewarding or punishing them to encourage or discourage certain long-term behaviors. RL methodologies have been successfully applied to a variety of tasks like robot control, both in simulated and real-world scenarios, and AIs for complex strategy games like Go, chess and others.

The key concept of RL is the interaction between the *agent* and the *environment*, which represents the world that the agent exists in and interacts with. At each step of the interaction, the agent observes the state of the world (or a part of it) and selects an *action* to take, then the environment changes in response to such action or possibly even on its own. Together with the *observation*, i.e., the possibly partial information about the state of the world perceivable by the agent, the agent also receives a *reward* value from the environment telling how good or bad the current state of the world is. The aim of the agent is to maximize the cumulative reward (also known as *return*) obtained by the environment.

The solution to an RL problem is a *policy* π, i.e., a mapping from states to actions that maximizes the expected return when the agent acts according to it. To compare different policies in terms of effectiveness, the concept of *value* is introduced: Informally, the value of a state under a given policy is the expected return if the agent starts in that state and then follows the policy from there on. A policy π is better than or equivalent to another policy π' if the value of π is greater than or equal to the value of π' for every state. The goal of RL

algorithms is thus to learn (approximate) the *optimal* policy, i.e., the policy which is better than or equivalent to every other policy. In our case, the optimal policy will be encoded as a fully connected ReLU network which provides a mapping between states and actions.

Given the employ of RL as a framework for conducting robotics-based studies of NN verification, in this work we confine ourselves to an informal description of reinforcement learning. For more details we refer to Plaat (2022); Sutton and Barto (1998).

2.3 Constraint Satisfaction Problems

A Constraint Satisfaction Problem (CSP) requires a value, selected from a given finite domain, to be assigned to each variable in the problem, so that all constraints relating the variables are satisfied Brailsford et al. (1999). In more detail, given a set of variables together with their domains, i.e., a set of possible values that can be assigned to each variable, and a description of the problem in the form of mathematical constraints, a CSP is the problem of finding values of the variables that satisfy every constraint. It is defined as a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of current *domains* $\mathscr{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible *values* for variable x_i, and a set $\mathscr{C}$ of *constraints* between variables. A constraint C on the set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset of the Cartesian product $D_0(x_{i_1}) \times \ldots \times D_0(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1} \times \ldots \times x_{i_r}$.

In this Thesis we consider two main approaches to solve CSPs, namely Constraint Programming (CP) and Satisfiability Modulo Theory (SMT). CP is a powerful paradigm for solving CSPs that draws on a wide range of techniques from artificial intelligence, operations research, algorithms, graph theory and others Rossi et al. (2006). In general, CP provides high level languages that allow one to describe (model) a problem in a declarative way by means of constraints, that is, properties of the solutions to be found. Product configuration has been established as a successful application area of CP Benavides et al. (2005); Falkner et al. (2011); Hervieu et al. (2016); Jensen (2004); Mcdonald and Prosser (2002). A CP solver checks whether a certain assignment of decision variables respects all the constraints, and returns such assignment in that case. Most CP solvers deal naturally with finite domains and have good propagators to handle injectivity constraints. SMT handles the original problem by encoding it to the problem of deciding the satisfiability of a first-order formula with respect to some decidable theory $\mathscr{T}$. In particular, SMT generalizes the Boolean satisfiability problem (SAT) by adding background theories such as the theory of real numbers, the theory of integers, and the theories of data structures (*e.g.*, lists, arrays and bit vectors) — see, e.g., Barrett et al. (2009) for details. There exists a constraint modeling language for designing constraint satisfaction and optimization problems in a high-level, solver independent way called MiniZinc Marriott et al. (2014). SMT solvers comply instead to a dedicated standard language called SMT-LIB Barret et al. (2017).

Here we provide some more detail about the SMT approach which we use as a basis to introduce our modeling of design and configuration for RHEs. The corresponding CP formulation can be obtained with a simple syntax-driven translation. To decide the satisfia-

bility of an input formula φ in conjunctive normal form, SMT solvers typically first build a *Boolean abstraction* $abs(\varphi)$ of φ by replacing each constraint by a fresh Boolean variable (proposition), e.g.,

$$\begin{array}{rl} \varphi & : \underbrace{x \geq y} \wedge (\ \underbrace{y > 0} \vee \underbrace{x > 0}\ ) \wedge \underbrace{y \leq 0} \\ abs(\varphi) : & \ A \ \wedge (\ \ B \ \ \vee \ \ C \ \ ) \wedge \ \neg B \end{array}$$

where x and y are real-valued variables, and A, B and C are propositions. A propositional logic solver searches for a satisfying assignment S for $abs(\varphi)$, e.g., $S(A) = 1$, $S(B) = 0$, $S(C) = 1$ for the above example. If no such assignment exists then the input formula φ is unsatisfiable. Otherwise, the consistency of the assignment in the underlying theory is checked by a *theory solver*. In our example, we check whether the set $\{x \geq y,\ y \leq 0,\ x > 0\}$ of linear inequalities is feasible, which is the case. If the constraints are consistent then a satisfying solution (*model*) is found for φ. Otherwise, the theory solver returns a theory lemma φ_E giving an *explanation* for the conflict, e.g., the negated conjunction some inconsistent input constraints. The explanation is used to refine the Boolean abstraction $abs(\varphi)$ to $abs(\varphi) \wedge abs(\varphi_E)$. These steps are iteratively executed until either a theory-consistent Boolean assignment is found, or no more Boolean satisfying assignments exist.

Adding theories of cost to SMT yields Optimization Modulo Theories (OMT), an extension that finds models to optimize given objectives through a combination of SMT and optimization procedures Sebastiani and Tomasi (2012). For example,

$$\begin{cases} \varphi : x \geq y \wedge (y > 0 \vee x > 0) \wedge y \leq 0 \\ \min_{x,y}(x + y) \end{cases}$$

requires all the constraints in φ to be satisfied and the additional cost $x + y$ to be minimized. Notice that OMT extends classical formulations in mathematical programming, e.g., linear programming or mixed integer linear programming, since it allows Boolean structure to be taken into account together with the optimization target. OMT solvers have been developed for several first-order theories like, e.g., those of linear arithmetic over the rationals (LRA) or the integers (LIA) or their combination ($LIRA$). In this work we mainly consider *quantifier free* theories in a mixed integer/rational domain — known as *QF_LIRA* in the literature Barrett and Tinelli (2018).

2.4 Genetic algorithms

Genetic Algorithms (GAs) are optimization procedures based on ideas borrowed from natural selection and evolution. Detailed descriptions of GAs are to be found, e.g., in Davis (1991). An application of GAs to a relatively simple automated configuration problem together with a comparison with other declarative techniques can be traced back to Falkner et al. (2011), and other earlier references leveraging GAs for automated product configuration can be found in Zhang (2014). A recent survey Slowik and Kwasnicka (2020) cites many different applications of GAs and other evolutionary algorithms to engineering problems, including automated configuration and optimization scenarios such as, e.g., optimization of solar array layouts Lv et al. (2017), load balancing in cargo ships Ramos et al. (2018), and building energy-efficient houses Ascione et al. (2016). For the purpose of this paper, it is sufficient to recall that GAs consider a *population* as a finite set P of potential solutions to the target optimization problem. Each individual $p \in P$ is characterized by a *genotype* comprised of *chromosomes*. As in nature, chromosomes define the individual and are the basis for the obtaining different individuals by "mating" procedures. The *fitness function* is a mapping $f : P \to \mathbb{R}$ which ranks the individuals according to a *fitness score*: the higher the chance of being a good solution, the higher the fitness score. Notice that GAs provide *unconstrained optimization* over the space of potential solutions. In order to take into account constraints, as our elevator design problem requires, the fitness function should contain one or more *loss factors* — see, e.g., Güngör (2022); Homaifar et al. (1994); Yeniay (2005) — which penalize the individual design when it violates specific constraints: in this way, hard constraints are turned into preferences about solutions. By shaping the loss factors adequately we are able to control how much getting closer to violating a constraint can be discouraged. GAs are initialized with a randomly chosen population P and then they seek to improve the initial choice by repeating the following steps:

1. the fitness $f(p)$ of each individual $p \in P$ is computed;
2. a set $M \subset P$ is extracted from P such that individuals in M have the highest fitness among those in P;
3. the individuals in M are subject to "mating" procedures such as *crossover*, or other evolutionary phenomena such as *mutation*: informally, crossover occurs when the genotypes of two individuals are split and recombined to form new ones bearing some chromosomes, i.e., common traits, from both their parents.

4. The result of the previous step is a population P' which might contain individuals fitter than those of the previous population P; in particular, the crossover operation attempts to combine the genes of fit individuals to produce fitter children, and mutation attempts to maintain diversity in a population of designs.

5. Population P' becomes the new population P and the search restarts from step (1) unless some *termination condition* occurs, e.g., the fitness of the fittest individuals did not change in the last k steps, or a fixed number of h generations has been produced, where k and h are user-controlled hyper-parameters.

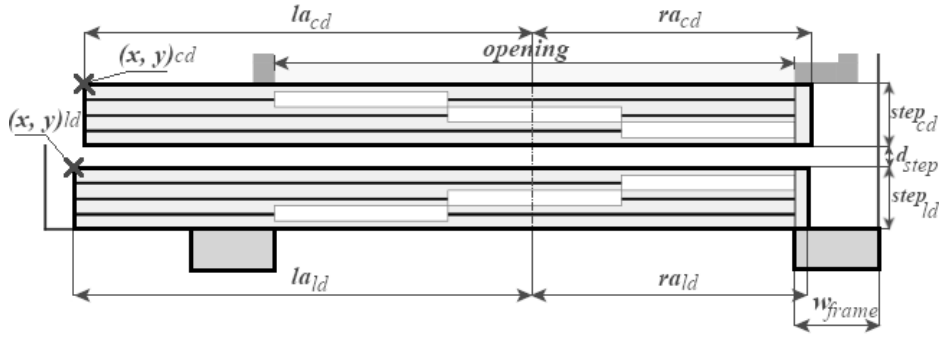
Part I

Automated System Design

Chapter 3

Design model

Figure 3.1 Detail of the car/landing door pair and related parameters.

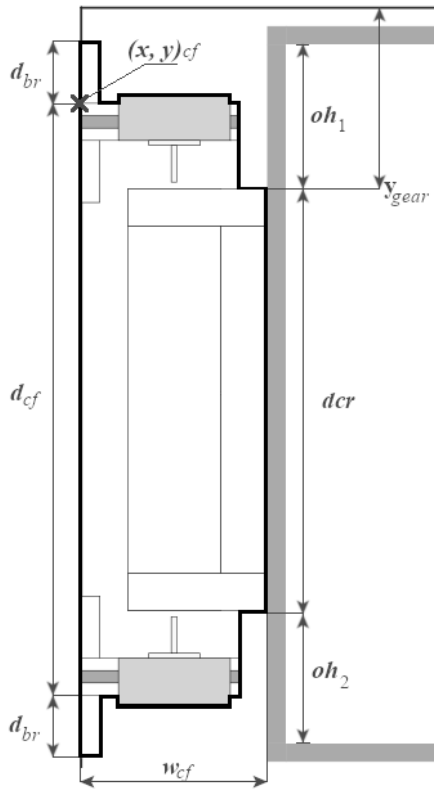


In this chapter we present our case study of the elevator system, how the model is translated into variables and parameters and which constraints we need in order to perform the configuration and the cost functions required to achieve design.

3.1 Variables and Parameters

We now introduce the decision variables and the parameters involved in the configuration task. We note that the reference system in LIFTCREATE origins from the top left corner of the internal shaft wall and the y axis is inverted with respect to a canonical Cartesian system. The origin $\mathcal{O}_{(x,y)}$ of this reference system coincides with the *shaft base point* (x_{shaft}, y_{shaft}) which is always set to $(0,0)$ — see Fig. 2.1. In Fig. 3.2 we present a fragment of the plan view focusing on the car frame structure, which is comprised of the *brackets* — wall-mounted T-shaped components — to support the car rails on which the car frame *core gear* slides. The *car frame base point*, i.e., the insertion point of the car frame structure in the configuration, lies on the outer corner of the topmost bracket and it is marked with a cross. The coordinates of the car frame base point (x_{cf}, y_{cf}) — denoted by $(x,y)_{cf}$ in Fig. 3.2 — determine a specific placement of the structure. The *overhang* of the car with respect to the car frame is the distance from the car walls to the car frame core gear edges — the top edge is y_{gear}. As shown in the drawing, the overhang correspond to two parameters oh_1 and oh_2, required to handle the cases in which the car is not centered with respect to the car frame. The parameter dcr denotes the distance between the car rails, i.e., the size of the core gear. Starting from the base point of the car frame, w_{cf} and d_{cf} are the width and the depth of the car frame, respectively, whereas d_{br} is the depth of the brackets; the total encumbrance of the car frame in the shaft is given by the sum $d_{cf} + 2d_{br}$.

Figure 3.2 Detail of the car frame and related parameters.



In Fig. 3.1 we consider a fragment of the plan view focusing on the door pair — notice that the car door and the landing door *opening* must be aligned. The drawing in Fig. 3.1 represents a pair of telescopic doors with 3 panels. The *car door base point* (x_{cd}, y_{cd}) — denoted as $(x,y)_{cd}$ in the plan view — and the *landing door base point* (x_{ld}, y_{ld}) — denoted as $(x,y)_{ld}$ in the plan view — are always at the top left corner of the corresponding structure. The value of these coordinates represents a specific placement of the car/landing door pair. The landing door opening is surrounded by the *frame*, i.e., the structure that surrounds the entrance to the car, with width w_{frame}. The total door width is the sum of two parameters, the *left axis* — la_{cd} and la_{ld} for car and landing door, respectively — and the *right axis* — ra_{cd} and ra_{ld} for car and landing door, respectively. Both axes originate from the opening midpoint and, as shown in the drawing of Fig. 3.1, in general they may not coincide. Finally, $step_{cd}$ and $step_{ld}$ denote the depth of the step in the car and landing door, respectively; d_{step} is the distance between car and landing doors.

In Tables 3.1 and 3.2 we summarize all the quantities involved in the configuration, separating decision variables (Table 3.1) from parameters (Table 3.2) either related to the initial specification or extracted from the components database — all quantities are in millimeters. We introduced all the decision variables beforehand with the exception of (x_{car}, y_{car}), i.e., the car base point coordinates corresponding to the top-left internal edge of the car in Fig. 2.1, and w_{car} and d_{car}, i.e., the width and depth of the car. Concerning parameters, we consider four groups of them. The first group is related to shaft measurements and includes w_{shaft} and d_{shaft} — width and depth of the shaft, respectively; also in this group we have *reductions* (red_N, red_S, etc.), i.e., the distance between the car walls and the shaft, and *car wall thicknesses* (cwt_N, cwt_S, etc.). For both such groups of parameters we have four values (N, S, W and E) to account for different sizes on all sides (top, bottom, left and right, respectively). The second group is related to car frame dimensioning and includes max_{oh},

Table 3.1 Explanation of the decision variables involved in the design process

Symbol	Description
x_{cf}, y_{cf}	Car frame base point coordinates
x_{cd}, y_{cd}	Car door base point coordinates
x_{ld}, y_{ld}	Landing door base point coordinates
x_{car}, y_{car}	Car base point coordinates
w_{car}, d_{car}	Car width and depth

i.e., the maximum overhang, and other parameters detailed in the table. The third group is related to doors — no further parameters need to be introduced in this group.

3.2 Constraints

Having defined our decision variables and parameters, we proceed to describe the (hard) constraints required to find feasible solutions, divided into two groups related to car frame and doors respectively. The constraints to place the car frame must take into account two main issues. First, given the shape of the brackets, it is not possible to model the car frame as a simple rectangle in order to fit it with the other components. Therefore the placement of the car frame is computed by subtracting residuals from the total shaft measures. Second, the placement of the car frame must take into account its maximum overhang, i.e., the car cannot "lean" too much outside the car frame core gear. The considerations above lead to the following set of constraints:

$$\begin{cases} y_{cf} - d_{br} \geq y_{shaft} \\ y_{cf} + d_{cf} + d_{br} \leq y_{shaft} + d_{shaft} \\ 0 \leq y_{cf} + y_{gear} - y_{car} < max_{oh} \\ 0 \leq y_{car} + d_{car} - y_{cf} - y_{gear} - dcr < max_{oh} \end{cases} \tag{3.1}$$

The first two constraints are required to fit the shape of the car frame, while the last two are required to satisfy the requirement about the maximum overhang.

Table 3.2 Explanation of the parameters involved in the design process

Symbol	Description
x_{shaft}, y_{shaft}	Shaft base point coordinates
w_{shaft}, d_{shaft}	Shaft width and depth
$red_{[N,E,S,W]}$	Distance between shaft and car walls (North, East, South, West)
$cwt_{[N,E,S,W]}$	Car wall thickness (North, East, South, West)
w_{cf}	Distance from x_{cf} to the left car wall
d_{cf}	External distance between car frame rails
y_{gear}	Core gear placement with respect to the car frame base point
d_{br}	External depth of the car frame bracket from the base point
dcr	Distance between car rails
max_{oh}	Maximum car overhang that the car frame is able to sustain
d_p	Diameter of the hydraulic cylinder barrell
$opening$	Doors opening
la_{cd}, ra_{cd}	Left and right axis size (car door)
la_{ld}, ra_{ld}	Left and right axis size (landing door)
$step_{cd}$	Car door step
$step_{ld}$	Landing door step
d_{step}	Distance between doors
w_{frame}	Landing door external frame width

The constraints to place the car/landing door pair should guarantee that both structures fit the shaft, that the actual opening fits the car and that the landing door frame does not exceed the shaft size. These requirements can be translated into the following set of constraints:

$$\begin{cases} x_{cd} \geq x_{shaft} \\ x_{ld} \geq x_{shaft} \\ x_{cd} + la_{cd} + ra_{cd} \leq x_{shaft} + w_{shaft} \\ x_{ld} + la_{ld} + ra_{ld} \leq x_{shaft} + w_{shaft} \\ x_{cd} + la_{cd} - \frac{opening}{2} \geq x_{car} \\ x_{cd} + la_{cd} + \frac{opening}{2} \leq x_{car} + w_{car} \\ x_{ld} + la_{ld} + \frac{opening}{2} + w_{frame} \leq x_{shaft} + w_{shaft} \end{cases} \tag{3.2}$$

The first four inequalities are required to guarantee that the car and the landing door structures fit the shaft; then we list two inequalities related to the car opening, and the last inequality guarantees that the landing door frame size is adequate for the shaft. In addition, the alignment of the landing door with respect to the car door must be enforced with the following equality constraint:

$$x_{ld} = x_{cd} + la_{cd} - la_{ld} \tag{3.3}$$

The placement of the car and landing door on the y axis is also enforced with equality constraints:

$$\begin{cases} y_{ld} = y_{shaft} + d_{shaft} - step_{ld} \\ y_{cd} = y_{ld} - d_{step} - step_{cd} \end{cases} \tag{3.4}$$

Further equality constraints are required to take into account that the door placement over the y axis, together with the car frame and door selection, influences the car size as follows:

$$\begin{cases} x_{car} = w_{cf} + cwt_W \\ y_{car} = red_N + cwt_N \\ w_{car} = w_{shaft} - w_{cf} - cwt_W - cwt_E - red_E \\ d_{car} = d_{shaft} - red_N - cwt_N - cwt_S - H_{doors} \end{cases} \tag{3.5}$$

where H_{doors} stands for the total door occupancy over the y axis computed as:

$$H_{doors} = step_{ld} + step_{cd} + d_{step} \tag{3.6}$$

Notice that when the car frame is positioned on the left hand side of the elevator, its x base coordinate x_{cf} is always set to 0.

Table 3.3 Grouping of the constraints involved in the design process, separated by their purpose

Class	Type	Number
Car Frame shape	*Inequality*	6
Doors shape	*Inequality*	7
	Equality	3
Car shape	*Equality*	4
Car Frame / Doors overlap	*Inequality*	2
Components selection (SMT)	*Implication*	308

Since the car door body may protrude over the car walls, in order to minimize the risk of collision with other components, designers must consider a safety margin. To guarantee this requirement, specific non-overlapping constraints are implemented. For example, if we let r be the security margin, the non-overlapping constraint relative to car frame and car door can be written as follows:

$$x_{cd} - r \geq x_{cf} + w_{cf} \vee y_{cd} - r \geq y_{cf} + d_{cf} \tag{3.7}$$

In Table 3.3 we summarize the shape of our problem by identifying the number of constraints and their type, based on their purpose. The last element of the Table is related to the SMT encoding where we use Boolean implications as constraints to choose the components.

3.3 Objective

In our process, we considered four design objectives that are the main interests for technical engineers. The first is that the car frame should be aligned as much as possible to the center of the car on the y axis — i.e., considering Figure 2.1, the car frame should appear vertically centered with respect to the car. The second objective regards doors positioning: in case of symmetric doors, i.e., the opening midpoint coincides with the door frame midpoint, the opening of the car door should be aligned as much as possible to the center of the car on the x axis; in case of asymmetric doors, then the opening should be as close as possible to the opposite side of the car frame in order to minimize the chance of interference. Notice that in Figure 2.1 we have a non-symmetric door, which is aligned to the right car wall. The third objective is that the car frame and the hydraulic cylinder should not be over-sized, and the

opening of the car door should not be undersized during the components selection phase and finally, the last objective is to minimize the force exerted by the elevator on the car rails.

Here we describe the single objectives and we shape them as contributions to a cost function, mentioning the details of the parameters involved when necessary. The cost associated to car frame misalignment on the y axis is expressed by the absolute value of the distance between the car frame and the car axes. We define the car frame vertical axis $axisY_{cf}$ as the car frame vertical base point y_{cf} plus half of the distance between the car rails dcr

$$axisY_{cf} = y_{cf} + \frac{dcr}{2}. \tag{3.8}$$

The vertical car axis $axisY_{car}$ is defined as the car vertical base point y_{car} plus half of the car depth d_{car}:

$$axisY_{car} = y_{car} + \frac{d_{car}}{2}. \tag{3.9}$$

The difference between the terms (3.8) and (3.9) gives us the first contribution to the cost function c_{cf}:

$$c_{cf} = |axisY_{cf} - axisY_{car}| \tag{3.10}$$

The second objective we consider is related to doors. In this case we define the horizontal car axis $axisX_{car}$ as the horizontal car base point x_{car} plus half of the car width w_{car}:

$$axisX_{car} = x_{car} + \frac{w_{car}}{2} \tag{3.11}$$

The horizontal door axis $axisX_{door}$ is defined as the horizontal door base coordinate x_{cd} plus the length of its left axis la_{cd}:

$$axisX_{door} = x_{cd} + la_{cd} \tag{3.12}$$

In the case of symmetric doors, good design practices suggest that $axisX_{door}$ and $axisX_{car}$ should be aligned. In the case of non-symmetric doors, it is preferable to have the door *opening* as close as possible to the side of the car which is opposite to the car frame. In a configuration like the one in Figure 2.1 we can define the base coordinate of such side as:

$$x_{wall} = x_{car} + w_{car} \tag{3.13}$$

To take into account the different arrangement of doors, we introduce a binary variable, δ_t, which is assigned to 1 if the current door is a non-symmetric door and to 0 otherwise. We

can then summarize the contribution to the cost function as:

$$\begin{aligned} c_{door} = (\quad & (1-\delta_t)|axisX_{car} - axisX_{door}| + \\ & \delta_t(x_{wall} - (axisX_{door} + \tfrac{opening}{2})) \quad) \end{aligned} \tag{3.14}$$

The first contribution of (3.14) is zero when $\delta_t = 1$, i.e, for non-symmetric doors we try to minimize the distance from the side of the elevator opposite to the car frame, whereas when $\delta_t = 0$ we try to align the door and the car axes. The third objective is related to the selection of the components, and gives the guidelines for sizing the car frame, the doors and the cylinder. The maximization of the door *opening* leads to accessible elevators which are always considered a plus, whenever feasible; the minimization of the car frame depth dcr and the barrel diameter d_p suggests components which are not over-sized, thus helping to keep costs at bay. These criteria can be translated into one additional contribution to the overall cost function defined as:

$$c_{size} = (dcr + d_p - opening) \tag{3.15}$$

Finally, the last term to minimize is the sum of F_{cr}^x and F_{cr}^y, i.e., the x and y components of the force exerted on the car rails F_{cr}:

$$c_{cr} = F_{cr}^x + F_{cr}^y \tag{3.16}$$

The computation of F_{cr} is non-trivial and requires additional equations and parameters that we briefly describe. The components of F_{cr} are obtained as

$$\begin{aligned} F_{cr}^x &= k \cdot g \cdot \tfrac{Q_x(Q+75)+P_x \cdot car_W + cdP_x \cdot cd_W + cf_W \cdot CF_x}{2 \cdot h} \\ F_{cr}^y &= k \cdot g \cdot \tfrac{Q_y(Q+75)+P_y \cdot car_W + cdP_y \cdot cd_W}{2 \cdot h} \end{aligned}$$

where the parameters have the following meaning:

- k is a parameter depending on the kind of safety brakes installed;
- g is the standard acceleration due to gravity;
- Q is the car payload;
- P_x and P_y are the midpoint coordinates of the car;

- Q_x and Q_y are obtained through the equations

$$\begin{aligned} Q_x &= max\{P_x + \tfrac{w_{car}}{8}, P_x - \tfrac{w_{car}}{8}\} \\ Q_y &= max\{P_y + \tfrac{d_{car}}{8}, P_y - \tfrac{d_{car}}{8}\}; \end{aligned}$$

- car_W is the car weight;
- cdP_x, cdP_y are the coordinates of the center of gravity of the car door;
- cd_W is the car door weight and cf_W is the car frame weight;
- CF_x is a coefficient computed as

$$CF_x = 1.5 \cdot \frac{w_{cf}}{2}$$

  where w_{cf} is the distance from the car frame base point to to the left car wall;

- h is the distance between guide shoes, i.e., the supports which slide on the car rails.

Chapter 4

Design encoding

In this chapter we present the encoding of the elevator constraints, and how different choices can impact the performance of the design task. First, we present the baseline heuristic method and some preliminary approaches dealing with Genetic Algorithms (GAs). Then, we elaborate on lessons learned in four main aspects of the encoding, both from a general point of view by discussing the employ of different kinds of constraints, and from a specific point of view by discussing how arithmetic and optimization techniques can handle the problem — leading to the CSP-based variations of LIFTCREATE.

In the following chapters, we use a number of variants of LIFTCREATE while referring to the algorithm considered:

- LIFTCREATE-HR is the current heuristic-based engine
- LIFTCREATE-BF is a brute-force version used as a baseline
- LIFTCREATE-GA is the Genetic Algorithm-based experiment
- LIFTCREATE-RS is a random-sampling version used as a baseline w.r.t. LIFTCREATE-GA
- LIFTCREATE-SMT is the SMT-based experiment
- LIFTCREATE-CP is the CP-based experiment

4.1 Encoding strategies

Due to the specific features of LIFTCREATE versions, there are some differences in how configurations are generated and results are presented, differences that we describe in the following and that we tried to harmonize as much as possible in order to make our experimental comparison meaningful. LIFTCREATE-HR produces at most one solution — supposedly the "best" one according to the heuristics — for each *prototype*, i.e., a pair comprised of a door and a car frame which together fit the shaft. Therefore, for each given setup, LIFTCREATE-HR produces as many solutions as there are prototypes for which a solution exists, where a solution features a specific placement of car frame and doors. The three versions of LIFTCREATE based on search in the space of configurations, namely LIFTCREATE-BF, LIFTCREATE-RS and LIFTCREATE-GA, feature a common data flow implementing the following phases:

- *Prototype generation*: amounts to list all prototypes, pruning up-front those which cannot fit the given shaft.

- *Expansion*: given a prototype, potential configurations are explored by attempting to place the car frame and the doors inside the shaft within the allowable ranges: the results of this phase are *early designs*.
- *Design validation*: given an early design, a number of checks is performed in order to validate the corresponding configuration and declare it feasible or not.

Specifically, the validation phase must guarantee that every constraint of Section 3.2 is satisfied. The difference among LIFTCREATE-BF, LIFTCREATE-RS and LIFTCREATE-GA lies in the expansion phase: exhaustive search for LIFTCREATE-BF, random sampling for LIFTCREATE-RS and genetic algorithms for LIFTCREATE-GA. Since these versions may produce many feasible configurations for each prototype, whereas LIFTCREATE-HR outputs only one, we *cluster* configurations after the validation phase. In more details, the set of valid placements for each given prototype is clustered around a representative, in order to make the comparison with LIFTCREATE-HR possible. Finally, both in LIFTCREATE-CP and LIFTCREATE-SMT the prototype expansion phase is replaced by the CP/SMT encoding of the constraints introduced in Section 3.2, where the choice of components is restricted to those that can fit the shaft. The subsequent phases of expansion and validation are merged in the search for a solution by the solvers. If a cost function to drive the search towards "optimal" designs is supplied, then LIFTCREATE-CP and LIFTCREATE-SMT output exactly one configuration for each given setup.

4.1.1 Custom heuristics: LIFTCREATE-HR

The special-purpose heuristic engine of LIFTCREATE-HR is meant to replicate the design flow that a professional would carry out by hand. Algorithm 1 is a pseudo-code representation of the function COMPUTE_DESIGNS which is at the core of LIFTCREATE-HR. COMPUTE_DESIGNS takes as input the shaft dimensions w_{shaft} and d_{shaft}, the list of all available car doors D and the list of all available car frames F. The car door list is sorted in *decreasing order* according to door size (ORDER_BY_OPENING) and the car frame list is sorted in *increasing order* according to the distance between car rails (ORDER_BY_DCR). The outermost **for** loop scans the available car frames, computing for each one the car surface S_{cur} (COMPUTE_CAR_SURFACE). Here, a heuristic choice is performed: if the car surface allowed by the current car frame S_{cur} is not greater than the one allowed by the previous one S_{prev}, the algorithm skips to the next choice of car frame. If S_{cur} is greater than S_{prev}, the search continues by filtering the list of car doors (FILTER_DOORS) to discard those larger than the shaft, while the doors left are scanned in the innermost **for** loop — according to

Algorithm 1 Heuristic algorithm for LIFTCREATE-HR

```
1:  function COMPUTE_DESIGNS(w_shaft, d_shaft, D = List<Door>, F = List<CarFrame>)
2:      output = [ ]                                          ▷ Design list, initially empty
3:      S_prev = 0                                            ▷ Car surface initialization
4:      ORDER_BY_DCR(F)
5:      ORDER_BY_OPENING(D)
6:      for i = 1 : F.length do
7:          S_cur = COMPUTE_CAR_SURFACE(F[i], w_shaft, d_shaft) ▷ Compute surface given the Car
    Frame
8:          if S_cur > S_prev then                            ▷ Search only if the surface increases
9:              S_prev = S_cur
10:             D = FILTER_DOORS (w_shaft, D)                 ▷ Discard doors that do not fit the shaft
11:             prototype = ∅                                 ▷ Empty prototype
12:             for j = 1 : D.length do
13:                 ld = MATCH_LANDING_DOOR(D[j])             ▷ Find a matching landing door for D[j]
14:                 prototype = (F[i], D[j], ld)
15:                 if IS_FEASIBLE(prototype) then            ▷ Check whether the tuple is feasible
16:                     BREAK
17:             if prototype ≠ ∅ then
18:                 design = ADD_RAILS(prototype)             ▷ Add car frame rails to the prototype
19:                 OPTIMIZE_COMPONENTS(design)               ▷ Align components
20:                 if IS_VALID(design) then                  ▷ Validate design after optimization
21:                     APPEND(output, design)
22:     return output
```

the order of D, the largest opening is considered first to maximize accessibility. For each door, a matching landing door is selected and a *design prototype* is created as the tuple $(F[i], D[j], ld)$. The predicate IS_FEASIBLE checks whether the prototype is feasible, i.e., the three components can be fitted in the shaft while respecting all the hard constraints. If so, the innermost loop comes to an end: implicitly, this is also a heuristic choice since the procedure does not check other doors that could provide alternative design prototypes. Given a prototype, the design is finalized by adding the car frame rails and optimizing the placement with the function OPTIMIZE_COMPONENTS. The predicate IS_VALID checks whether the design is valid and should be retained for further processing. Summing up, the procedure produces *at most* one design per prototype associated with a specific car frame. It is important to notice that some combinations of car frame and doors that may result in feasible designs are never explored. While this helps in keeping the search space at bay, LIFTCREATE-HR may return no designs also in scenarios where feasible ones exist.

4.1.2 Genetic algorithms: LIFTCREATE-GA

In LIFTCREATE-GA the genotype is composed by 4 chromosomes, each one consisting of a single gene. As in Carlson (1996), genes are represented by integer numbers as follows:

- *Gene 1*: Value of the car door x base point — x_{cd} in Table 3.1.
- *Gene 2*: Value of the car frame y base point — y_{cf} in Table 3.1.
- *Gene 3*: Choice of the car frame; to encode the choice among available car frames we assigned to each one a unique integer identifier.
- *Gene 4*: Choice of the car door; also in this case we encoded each door with a unique integer code.

Since the available car frame and door identifiers, i.e., the domains of the genes 3 and 4, are restricted to those that could fit a given shaft, each individual represents a single early design. We do not encode the variables x_{cf} and y_{cd} because, as pointed out in Section 3.2, the coordinates are fixed in the design.

The fitness function to score individuals is computed by associating costs corresponding to violations of the feasibility constraints, i.e., we turn hard constraints into soft ones to discourage designs that violate them. However, we cannot completely exclude unfeasible designs and this is why LIFTCREATE-GA still retains a validation step at the end to make sure that all generated designs are valid. The cost function built in LIFTCREATE-GA includes also terms encoding the objective presented in Section 3.3 for the choice and placement of the car frame and the doors. We penalize projects in which the car frame is misaligned with respect to the car axis — such as in (3.10). If the choice of the car frame is such that overhangs are negative — i.e., if the car frame dcr is greater than the resulting car depth d_{car} — the resulting value is multiplied by 10^3. Concerning doors, the cost detailed in (3.14) is applied depending on the door type in order to discourage misaligned placements while for the selection we apply a slight variation to the hard minimization of (3.15) by using a parabolic function that grows up quickly for values less than 550mm or greater than 800mm. The function we consider is $f(x) = 0.01 \cdot x^2 - 13.5 \cdot x + 4561.25$ computed with $x = opening$. This is because doors whose opening is less than 550mm and doors with opening greater than 800mm are discouraged: the former are used only in very special situations and the latter are usually very expensive. Whether the selected door is out of range, we apply a fixed weight which is either 10^5, if the actual door width is less than 550mm, or 300 if greater than 800mm. We use a parabolic function because, as we mentioned before, GAs provide

unconstrained optimization over the solution space, and a continuous objective is better for the fitness computation. The GA implementation is provided by the *Jenetics* Wilhelmstötter (2022) Java library, which allowed us to seamlessly plug the GA engine to our codebase by specifying the composition of the genotype.

4.2 Constraint satisfaction: LIFTCREATE-CP/SMT

LIFTCREATE-CP and LIFTCREATE-SMT implement each with their own specific formalism the encoding detailed in Chapter 3. Nevertheless, while experimenting in early stages of this encoding and guided by the experience obtained with search-based techniques we identified the following encoding alternatives that can impact the performance of a constraint solver.

4.2.1 Component selection

The car frame, the cylinder and the doors are selected from a database of components. In order to automate the design of an elevator, we must consider that choosing different components yields different parameter values for each one. The relationship between the selection of a component and the assignment of the corresponding parameter values can be encoded via Boolean implications of the form

$$Id_x = i \Rightarrow x.p = v \tag{4.1}$$

where Id_x encodes the identifier of choice for component x (a decision variable), i is a specific identifier value, $x.p$ is some parameter of the component x and v is the value of $x.p$ given that the component x with identifier i was chosen — see, e.g., Bacchus (2007). To encode constraints of the form (4.1) a combination of Boolean reasoning with integer arithmetic is sufficient. However, considering the way data sets are usually encoded in MiniZinc with arrays Marriott et al. (2014), we consider an alternative encoding where we associate an array to each component parameter. For example, if a component x has two parameters p_1 and p_2, we build two arrays P_1 and P_2 that will store the values of p_1 and p_2 for each instance of the component. The index of the arrays becomes a decision variable chosen by the solver to enforce the correct values of the parameters.

Table 4.1 Look-up table to encode the number of passengers P with starting value P_0 as a function of the car surface A.

Car surface (A)	**No. of passengers** (P)
$a1 < A \leq a2$	P_0
$a2 < A \leq a3$	$P_0 + 1$
$A > a3$	$P_0 + 2$

4.2.2 Look-up tables

Some parameters, e.g., the maximum number of passengers that the car may accommodate, are a function of others, e.g., the car surface. However, instead of expressing such constraints directly — which might involve the use of non-linear or transcendental functions — the correspondence between free parameters and derived ones is encoded with *look-up tables*. Table 4.1 exemplifies such a table assuming that the car surface A is contained within three ranges.

The car payload is computed in a similar way, but, since the surface ranges are different, we need another set of constraints structured in the same way. These requirements can be easily modeled with implications in the same way as component selection: the surface A is a decision variable that implies the number of passengers or the payload. However, both SMT-LIB and MiniZinc allow users to define custom *functions*. In practice, functions are series of *if-then-else* statements about, e.g., the car surface, where each function returns, e.g., the corresponding number of passengers or the payload.

4.2.3 Integers vs. Reals

Most parameters involved in the design process for elevators are expressed in millimeters which suggests integer-based encodings. However, some parameters, like the forces exerted on the car rails, involve arithmetic over reals. This makes the corresponding constraint satisfaction problems members of the mixed-integer arithmetic family. In such encodings, the main disadvantage is that a large number of integer quantities may increase considerably the solution time. We try to improve on this by relaxing some of the integer quantities to reals. In particular, we consider component parameters since parameters are not *decided* but their values are only assigned based on the choice of a component. This means that the domain of the parameters is a finite set and we can relax the arithmetic encoding without producing invalid results. In this representation the only operation that could add decimal digits is division, but since in our encoding there are only a few such operations, boundary

checking can be implemented easily. These considerations do not hold for some decision variables including, but not limited to, the index used to select components. Also, several CP solvers we experimented with are not affected by this choice due to the fact that they do not support floating-point arithmetic.

4.2.4 Single and Multi-objective optimization

Here we describe alternative constructions of the cost function, using the contributions that are detailed in Section 3.3. In previous works of ours Demarchi et al. (2019) we consider the weighted sum of the costs c_{cf}, c_{door} and c_{size} to obtain the overall cost function, but the contribution c_{cr} may conflict with the previous ones because the farthest is the door from the car frame, the greater is the force exerted on the car rails Cicala et al. (2022); Demarchi et al. (2021). Nevertheless, since the car rails can be chosen once the other components are fitted, this objective can be considered with a lower priority. If we follow a single-objective approach, we can weight the cost c_{cr} significantly less than the other three. The overall cost function $\mathscr{C}$ becomes

$$\mathscr{C} = \alpha_1 c_{cf} + \alpha_2 c_{door} + \alpha_3 c_{size} + \alpha_4 c_{cr} \tag{4.2}$$

with $\alpha_4 \ll \alpha_i$ for $i \neq 4$.

Alternatively, we can exploit priorities among different cost functions by resorting to multi-objective optimization using, e.g., the lexicographic method whereby preferences are imposed by ordering the objective functions according to their significance — see Chang (2015) for details. In this case, we consider two different cost functions:

$$\begin{aligned} \mathscr{C}_1 &= \alpha_1 c_{cf} + \alpha_2 c_{door} + \alpha_3 c_{size} \\ \mathscr{C}_2 &= c_{cr} \end{aligned} \tag{4.3}$$

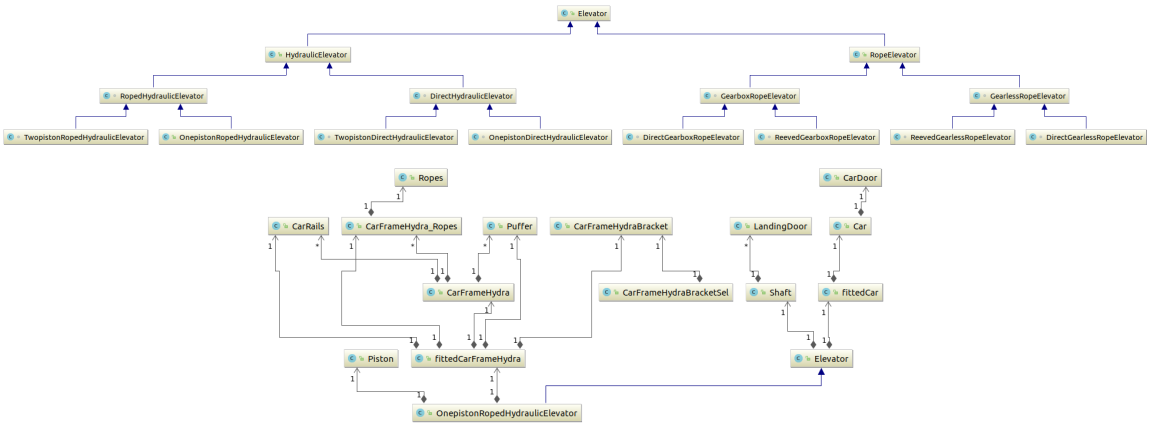
where the objective function $\mathscr{C}_1$ is minimized first.

LIFTCREATE-CP and LIFTCREATE-SMT take into account the constraints and the cost function providing an encoding to a formula in the MiniZinc Nethercote et al. (2007) and SMT-LIB Barret et al. (2017) languages, respectively, to be solved by a number of state-of-the-art solvers.

Chapter 5

LIFTCREATE

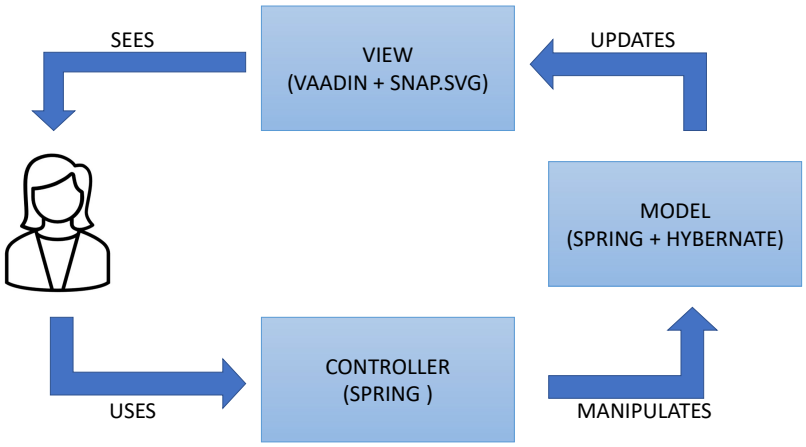
Figure 5.1 Taxonomy of LIFTCREATE's elevator models (top) and details of the components of `OnePistonRopedHydraulicElevator` (bottom). Rectangles represent entities, IS-A relations are denoted by solid arrows, and HAS-A relations are denoted by diamond-based arrows.



In this chapter we describe our tool LIFTCREATE, which implements the heuristic algorithm presented in the previous sections. When considering the new approaches for their evaluation we shared the same encompassing environment and interface. LIFTCREATE is a web-based application written in Java. To create designs from specifications, the tool retrieves all the components data from a database of commercial parts from different suppliers and explores the space of potential solutions guided by the user's preferences. The interaction with the user allows to shape and refine the heuristic search and pruning in the solutions space: the heuristic procedure detailed in Algorithm 1 is geared towards producing solutions as close as possible to human-conceived ones.

In practice, LIFTCREATE takes the designer from the very first measurements and requirements, e.g., shaft size and payload, to a complete design which guarantees feasibility within a specific normative framework. To achieve this, as first the user is asked to enter relevant parameters characterizing the project and an overall goal to pursue. For instance, if the size of the elevator's shaft is known and fixed in advance, LIFTCREATE can generate solutions which maximize payload, door size or car size. The goal is just a set of guidelines which, e.g., prioritize door size over other elements still keeping into account the hard constraints.

Figure 5.2 Software architecture schema of LIFTCREATE. The three main modules reflect the Model-View-Controller pattern and are connected via REST calls.



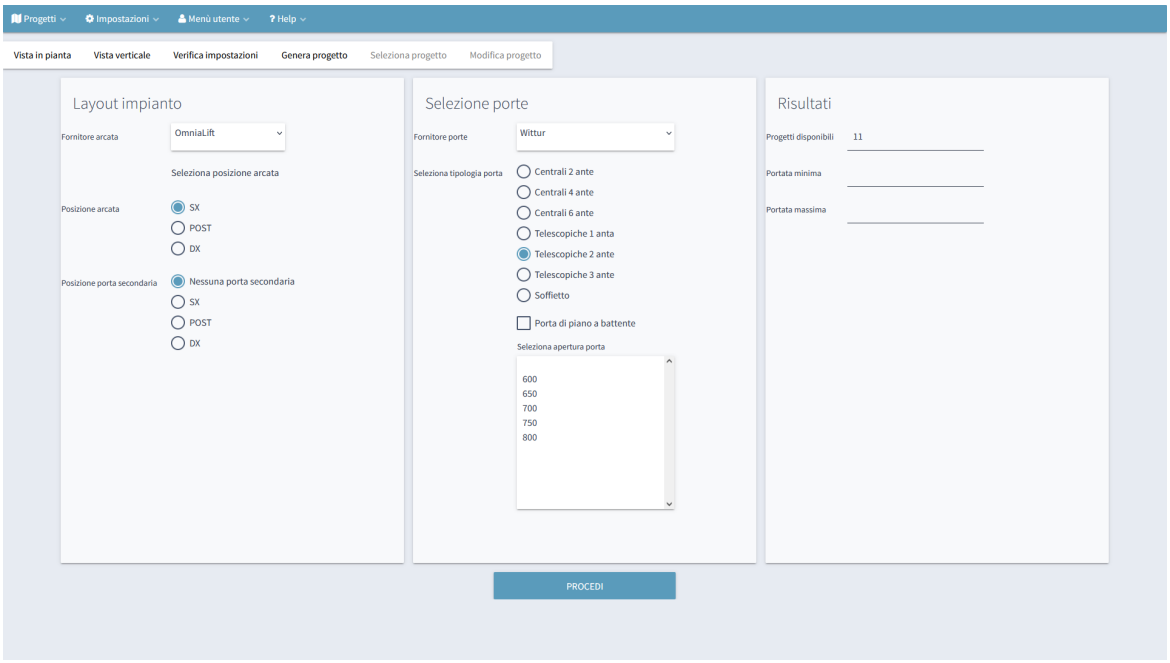
5.1 Software architecture

In order to manage the space of potential designs, LIFTCREATE does not consider the RHEs components detailed in Section 2.1 solely as drawing elements, but they must be handled as first class data inside the application logic. For example, `OnePistonRopedHydraulicElevator` is both a leaf in the taxonomy shown in Figure 5.1 (top) and also the root node of the corresponding part-whole hierarchy (bottom).

Looking at the hierarchy, the structure of RHEs with one piston direct drive can be easily learned, the only peculiar aspect being that these implements feature only one piston (`Piston`). The remaining components are common to all `HydraulicElevator` or `Elevator`. In particular, the car frame (`CarFrameHydra`), i.e., the mechanical assembly connecting the car with the piston, is specific of hydraulics-powered elevators. Albeit not physically part of the car frame, the entities `CarRails`, i.e., the rails along which the car is constrained to move, `Buffer`, i.e., the dumping device placed at the bottom of the elevator shaft, and `Ropes`, are logically part of it since their type and size must be inferred from or melded with the type and size of the car frame.

Common to all elevator types, the entities `Shaft` and `Car` are both logically part of the `Elevator` entity, but only the `Car` is also a physical component together with its sub-

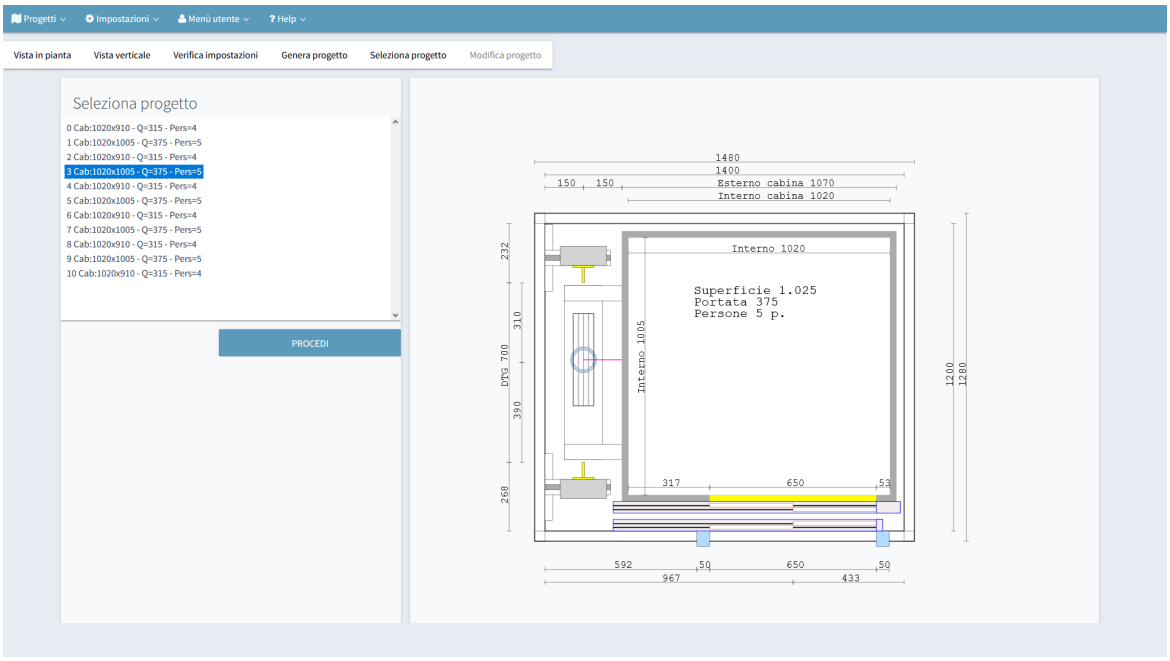
Figure 5.3 Screenshot of LIFTCREATE's guidelines for generating designs. It is possible to select the vendor for the car frame mechanics as well as for the doors, and a further filter distinguishes between different door families.



component `CarDoor`. In the case of the `Shaft`, while landing doors (`LandingDoor`) are not physically part of it, they are attached to it and their size and type must be inferred from or melded with car doors. The relationships encoded in such part-whole hierarchy are instrumental to LIFTCREATE when it comes to handle drawing, storage and retrieval of designs, but also to reason about the various trade-offs of a design while searching in the space of potential solutions.

The heuristic configuration engine and the other approaches are developed as REST services (REpresentational State Transfer), i.e., they exchange calls and information with the client browser as JSON objects via HTTP. In this way, it is possible to call a specific method by mapping an URL address supporting GET and POST arguments. In Figure 5.2 the schematic architecture following the Model-View-Controller (MVC) pattern is depicted. It divides the related program logic in three interconnected elements and separates the internal representations of information from the user's point of view. The software is deployed server-side using the *SPRING* Java framework, which is an asynchronous non-blocking architecture for developing and deploying web applications in a fast and secure way. We use this framework alongside with *Hybernate*, an Object Relational Mapping resource for

Figure 5.4 Screenshot of a LIFTCREATE design obtained in the web application. On the left it is possible to see the alternative designs proposed.



achieving object persistence in the application and easily interchange the object representation between SQL and Java.

5.2 Web interface

In order to build an interface which is fluent and interactive with the underlying logic, we build on the Java backend running the engine with the VAADIN framework The Vaadin Team (2019). VAADIN allows us to incorporate the web interface in the same Java project, keeping available all the logic components and easing interaction between the user and the engine.

The web interface of LIFTCREATE is visible in Figures 5.3 and 5.4. Figure 5.3 is a screenshot of the heuristic engine setup: here the user preferences are arranged in two areas. First, the user chooses the elevator layout by selecting the car frame vendor (*Fornitore arcata*) and its position in the design (*Posizione arcata*) — the default position is on the left, and the main door is always at the bottom. Here the user can also request the placement of a second door (*Posizione porta secondaria*). Once the layout is chosen, the door selection tab allows to further filter the solutions space by selecting *i*) the door vendor (*Fornitore porte*), *ii*) the door type (*Seleziona tipologia porta*) between symmetric, telescopic and folding with

a different number of panels and *iii*) the door opening. This selection is actually evaluated in real time given the car frame and shaft measurements, filtering doors having an opening too big for the actual space.

In Figure 5.4 we show the result of an example configuration process, i.e., the output of the heuristic design generation. Once given the initial setting, LIFTCREATE presents a number of alternative designs — 11, in this case — matching the specifications. This is an extra feature that optimal approaches lack in principle, since they are designed to produce the single, best solution. The design is rendered in SVG and can be exported in the main formats for CAD users. The SVG includes all components and their dimensioning in order to obtain a full-fledged CAD design. We manage to render the design — and parts of it — in the browser thanks to the Snap.svg Adobe (2022) JavaScript library, that we use for wrapping custom components and arrange them in the viewport. Note that Figure 2.1 is a simplified and clean version of the plan view generated by LIFTCREATE where we erased the dimensioning. The top menu allows to fine-tune the general settings — *Impostazioni* — by specifying the car wall thickness, the reductions, i.e., the distance between the car walls and the shaft, and the safety margins.

Chapter 6

Experimental analysis

In this chapter we present our experimental analysis to understand the impact of different choices for encoding our problem. We recall that LIFTCREATE-HR, detailed in Algorithm 1, produces at most one solution per *prototype*, i.e., a pair of car frame and door. LIFTCREATE-BF, LIFTCREATE-RS and LIFTCREATE-GA share a common flow that (i) generates prototypes, (ii) expands design candidates for each prototype and (iii) validates early designs on the hard constraints of Section 3.2. Finally, both LIFTCREATE-CP and LIFTCREATE-SMT provide a direct encoding to a MiniZinc or SMT-LIB formula which is fed to the corresponding solvers.

6.1 Experimental setup

The results are obtained by running the different versions of LIFTCREATE using the same database to configure RHEs in sixteen different setups. In particular, a database of commercial components is considered for car frames and car doors. In the LIFTCREATE web application, the database collects all the components described beforehand as well as some additional support tables that are not considered here. The component tables have as many columns as the parameters that are depicted in Table 3.2, and the tables are built referencing actual measurements from a few part suppliers. When LIFTCREATE is required to choose a car frame and a car door, it queries the database and then initializes the specific procedures with such data. The results herewith presented consider a database consisting of 25 car frames, 236 doors and 47 hydraulic cylinders. We run the different versions of LIFTCREATE in sixteen different setups, i.e., configuration scenarios, including both cases in which, given the available components, feasible solutions exist, and others for which there are none. The setups we consider represent typical shaft sizes found in residential buildings: two families of 8 setups, the former featuring 1300mm shaft width and the latter featuring 1500mm shaft width; shaft depth varies in both families from 800mm to 1500mm. Overall, these setups enable a thorough evaluation of LIFTCREATE versions considering realistic settings. All tests run on a PC equipped with an Intel® Core™ i7-6500U dual core CPU @ 2.50GHz, featuring 8GB of RAM and running Ubuntu Linux 16.04 LTS 64 bit.

Considering the evaluation parameters, we always measure the run time since the main objective is to deploy an alternative, declarative-based encoding in order to replace the current heuristic engine. When considering LIFTCREATE-GA we also measure some statistical parameters by running 50 samples with a unique seed for each sample. Other measures, e.g., size of the explored search space, number of sub-problems generated by the CSP approaches, might make sense only for specific situations. We also consider two different

Table 6.1 Results of computing configurations with heuristic techniques (LIFTCREATE-HR) on the baseline encoding: "**Time**" is the total runtime in milliseconds, "**No. of configs.**" is the total number of feasible configurations found (at most one for each prototype).

Shaft size	Time	No. of configs.
1300×800	1271	0
1300×900	731	54
1300×1000	715	51
1300×1100	633	89
1300×1200	633	168
1300×1300	764	397
1300×1400	1062	679
1300×1500	966	859
1500×800	1213	0
1500×900	1250	80
1500×1000	1330	160
1500×1100	1268	198
1500×1200	1544	414
1500×1300	1742	920
1500×1400	1823	1179
1500×1500	1548	986

sets of experiments: a baseline encoding dealing with the configuration of the car frame and the door pair only, and a full encoding dealing also with the selection and sizing of the hydraulic cylinder as well as the minimization of forces on the car rails. In particular, in the baseline encoding we consider only the cost components related to car frame and doors, whereas the full encoding takes into account all the cost components. LIFTCREATE-GA considers only the baseline encoding, as well as LIFTCREATE-CP, while LIFTCREATE-SMT is compared on both encodings due to numerical stability reasons detailed in the next Section. LIFTCREATE-HR is tuned to be compared on both encodings.

When building the cost function, in the single-objective case, considering equation (4.2), we set the free parameters α_1, α_2 and α_3 to 0.3 and α_4 to 0.1 in order to encode different priorities. In the multi-objective encoding, we set all weights to one.

6.2 Experimental results

LIFTCREATE-HR. The results of LIFTCREATE-HR on the baseline encoding are reported in Table 6.1; as the results show, all the setups can be solved in less than 2 CPU seconds. Note

that the heuristic search herewith considered focuses only on the car frame and doors coupling, so that the results of the full design may appear inconsistent with Demarchi et al. (2019). The number of configurations found ranges from 0 for the two setups having shaft depth 800mm, to more than one thousand for deeper shafts. Notice that the number of configurations found by LIFTCREATE-HR may not coincide with the total number of feasible configurations: this is because heuristics in LIFTCREATE are geared towards providing arrangements that a human designer finds satisfactory and not just feasible ones.

To better appreciate the complexity of the configuration task and the results obtained with LIFTCREATE-HR, in Cicala et al. (2022) we present also the results obtained with LIFTCREATE-BF. In these experiments we follow the schema depicted in Chapter 4 where both the prototypes generation and the early designs expansion are performed by a brute-force search, i.e., for each prototype we produce a new early design by assigning both x_{cd} and y_{cf} to a possible value in their range. The runtime of LIFTCREATE-BF grows with the shaft size and it is up to three orders of magnitude greater than LIFTCREATE-HR, the largest slice of runtime consumed by the validation phase. This is reasonable because even if the expansion phase generates a large number of alternatives, no processing is performed on them. This means that LIFTCREATE-BF wastes a lot of processing time just to discard unfeasible configurations.

LIFTCREATE-GA. Table 6.2 shows the results of LIFTCREATE-GA. According to some preliminary experiments that we do not show for the sake of brevity, we set the mutation rate to 10%, i.e., individuals are affected by random gene mutation with a probability of 10%. In order to account for stochastic variability, we consider for each setting 50 sample runs of LIFTCREATE-GA, with a unique seed for each sample. In our implementation, the GA stops when the fitness of the fittest individuals remains unchanged for 10 generations in a row. Column "POP" refers to the population size, which is one tenth of the total solution space — overestimated as the product between the cardinalities of the domains of car frames and doors. Since we run different samples, we compute the median and interquartile range for each reported value: "V" is the number of valid designs and "C" is the number of what we defined in Section 4.1 clusters, i.e., designs sharing the same components. We also report the cardinality (median and interquartile range) of the intersection between LIFTCREATE-GA clusters and LIFTCREATE-HR and the ratio between the said intersection and the total number of clusters (column $\frac{\text{MED}_I}{\text{MED}_C}$ in Table 6.2). We can observe that, while the median of valid designs grows with the size of the shaft, their spread is almost always at least two orders of magnitude smaller. Concerning clusters, the difference between valid designs

Table 6.2 Results obtained with LIFTCREATE-GA with mutation value 10% on the baseline encoding. "**POP**" is the population size, "**V**" is the number of feasible projects, "**C**" is the number of clusters, "**I=C ∩ H**" is the number of clusters shared by LIFTCREATE-GA and LIFTCREATE-HR. For each pair, column "MED_x" and "IQR_x" are the median and the interquartile range of value x, respectively. Column "$\frac{\text{MED}_I}{\text{MED}_C}$" is the ratio between shared clusters and LIFTCREATE-GA ones.

Shaft size	**POP**	**V**		**C**		**I=C ∩ H**		$\frac{\text{MED}_I}{\text{MED}_C}[\%]$
		MED_V	IQR_V	MED_C	IQR_C	MED_I	IQR_I	
1300 × 800	0	0.0	0.0	0.0	0.0	0.0	0.0	–
1300 × 900	6392	5569.5	11.75	26.0	2.75	26.0	2.75	100.00
1300 × 1000	13192	11098.0	7.5	17.0	0.0	16.0	0.0	94.12
1300 × 1100	19992	16090.0	76.25	4.0	1.0	4.0	1.0	100.00
1300 × 1200	26792	22122.5	15.75	30.0	1.0	30.0	1.0	100.00
1300 × 1300	33592	26768.0	58.0	30.0	1.75	29.0	1.75	96.67
1300 × 1400	40392	32153.5	726.75	123.0	18.75	123.0	18.75	100.00
1300 × 1500	47192	37444.5	590.0	89.0	7.75	89.0	7.75	100.00
1500 × 800	0	0.0	0.0	0.0	0.0	0.0	0.0	–
1500 × 900	8272	1363.0	149.5	16.0	5.0	12.5	4.75	78.13
1500 × 1000	17072	7694.0	1811.75	48.0	4.75	47.0	5.5	97.92
1500 × 1100	25872	21195.5	48.75	52.0	0.0	52.0	0.0	100.00
1500 × 1200	34672	28658.0	58.0	66.0	1.75	59.0	1.75	89.39
1500 × 1300	43472	35697.0	30.5	89.0	2.0	89.0	2.0	100.00
1500 × 1400	52272	38497.5	1846.5	223.5	16.5	223.5	16.5	100.00
1500 × 1500	61072	47797.5	710.25	107.0	6.0	107.0	6.0	100.00

and their clusterization is also about two orders of magnitude, meaning that lots of feasible configurations share the same car frame and doors. Finally, the overlap of the clusters with the designs generated by LIFTCREATE-HR is substantial, reaching 100% on 9 out of 14 setups — the two setups in which no feasible design is found are not considered — and on the remaining setups it never drops below 78.13%. Overall, these results show that LIFTCREATE-GA is able to reach the same "quality" as LIFTCREATE-HR, provided a proper hyperparameter tuning.

In terms of sheer performances LIFTCREATE-HR is still superior to LIFTCREATE-GA, but the runtimes of the latter are reasonable for online deployment with the added flexibility of a "declarative" encoding: adding a new constraint to LIFTCREATE-GA only amounts

Table 6.3 Comparison of computation time for solvers on the baseline encoding: the first column reports the setup and the other columns report the time (ms) taken to solve each setup by the solvers — best times appear in boldface.

Shaft size	**MiniZinc**					**SMT-LIB**	
	OR-Tools	Chuffed	ECL^{i}PS^e	CPLEX	Gurobi	z3	OptiMathSat
1300×800	662	200	924	856	886	**100**	254
1300×900	645	**198**	703	1020	1802	432	30680
1300×1000	674	**192**	1401	918	933	416	58066
1300×1100	659	**179**	1734	940	971	582	154739
1300×1200	655	**191**	1796	1056	1237	417	82698
1300×1300	661	**188**	1771	1090	1725	495	100822
1300×1400	637	**188**	1366	918	887*	435	79323
1300×1500	672	**206**	875	1118	925*	517	98355
1500×800	644	199	678	1023	824	**116**	247
1500×900	664	**179**	691	902	881	787	101458
1500×1000	673	**195**	1379	987	887*	619	70082
1500×1100	639	**206**	1942	971	903	682	105071
1500×1200	660	**264**	2024	1060	934	501	83719
1500×1300	636	**224**	2412	987	1018	417	121801
1500×1400	645	**192**	1509	871	919	470	97753
1500×1500	653	**216**	845	856	935	463	142557

to add a term to the fitness function, whereas in the case of LIFTCREATE-HR any change involves modifying the code.

LIFTCREATE-CP and LIFTCREATE-SMT. We test our SMT-LIB encoding with two OMT solvers, namely z3 de Moura and Bjørner (2008) by Microsoft Research and OptiMathSat Sebastiani and Trentin (2018) by FBK, and our MiniZinc encoding with five different solvers. We use the lazy clause generation based solver Chuffed Chu (2013), the MiniZinc challenge winner Google OR-Tools Perron and Furnon (2020), the CLP solver ECL^{i}PS^e Schimpf and Shen (2012) and the two MIP solvers CPLEX IBM (2017) and Gurobi Gu et al. (2019). With all these solvers we can observe how different approaches in solving combinatorial optimization problems behave with our encoding choices. In more detail, we run Chuffed v0.10.3, OR-Tools v7.8, ECL^{i}PS^e v7.0, CPLEX v12.7, Gurobi v9.0.1, z3 v4.8.7 and OptiMathSat v1.7.0.1. We consider the default configuration of every solver, even if we are aware that tuning each solver for the specific problem might yield better results. However, we do not wish to introduce bias in our experiments due to the fact that we

may know a solver or a technique better than others and thus obtain effective configurations on specific solvers only. Notice that z3 and OptiMathSat do not generate proofs of their results in their default configuration. Overall, the baseline encoding features 29 parameters and 10 decision variables, whereas the full encoding features 42 parameters and 17 decision variables. The number of constraints varies from a minimum of 30 for the baseline encoding considering arrays and functions to 401 for the full encoding with implications to represent parameters and look-up tables.

In Table 6.3 we show the results obtained on the baseline encoding by all the solvers we consider. For each solver we report the best time obtained on two variations: one in which the selection of components is based on arrays and another featuring Boolean implications. Both variations are integer-based because not all the solvers support arithmetic over reals, so we do not consider relaxations here; also, since the car surface computation involves a division, we omit the deduction of the car payload and passengers which are required for a complete design. All the solvers leveraging MiniZinc encodings fare the best runtime when the component parameters are encoded with arrays: CP solvers like Chuffed seem to make effective use of element constraints and MIP solvers appear to handle the translation of array constraints better than Boolean implications. On the other hand, OMT solvers run faster on the version based on Boolean implications, as the addition of arrays involves dealing with more theories at once and this inevitably hurts performances.

As we can observe in Table 6.3, Chuffed is the one yielding the best runtimes, except for two setups where z3 is the fastest solver. Noticeably, these setups do not admit a feasible configuration given the shaft size and the components available. z3 and OR-Tools are second best, their runtimes being always less than one second; MIP solvers CPLEX and Gurobi seem slightly less effective than the leading pack. In some cases, marked with an asterisk in Table 6.3, Gurobi returned "*UNSAT or UNKNOWN*" as an answer even if a solution exists and the MiniZinc file is the same for all solvers, so we conjecture that numerical stability might be an issue in these cases, but we are investigating other potential causes. ECL^{i}PS^e results are mixed, i.e., some setups are solved faster than OR-Tools or z3, others take more than two seconds to solve. OptiMathSat is surprisingly slow on these encodings: if we exclude scenarios for which no feasible configuration exists, then OptiMathSat best result is 30 seconds to solve the 1300×900 setup.

When considering the full encoding, we limit our comparison to z3 and OptiMathSat, since they are the only ones that appear to handle encodings which contain a substantial part of arithmetic over reals involved in cylinder selection, sizing and computation of forces on the car rails. Among the MiniZinc-based tools, ECL^{i}PS^e is meant to support arithmetic over

Table 6.4 Comparison of computation time for z3 and OptiMathSat on the full encoding: the first column reports each setup; the other columns, grouped by solver, report runtimes (ms) of different versions: integer-based "**I**", relaxed "**I + R**" and relaxed with functions "**I + R + F**", respectively. Subcolumns "**SO**" and "**MO**" refer to single objective and multiobjective optimization, respectively — best times among z3 and OptiMathSat appear in boldface. The last column reports LIFTCREATE heuristic engine runtimes.

Shaft size	z3						OptiMathSat						Heuristic
	I		I + R		I + R + F		I		I + R		I + R + F		
	SO	MO	SO	MO	SO	MO	SO	MO	SO	MO	SO	MO	
1300 × 800	157	149	131	134	143	221	109	119	**100**	131	116	110	583
1300 × 900	8878	229522	1205	**844**	1978	2321	—	—	74960	53535	68215	52068	1784
1300 × 1000	36120	133325	2818	3362	4433	**2704**	156761	48328	136109	107356	132166	112379	921
1300 × 1100	36448	60589	3514	1967	2365	**1554**	192198	127753	160883	176491	199352	113889	2177
1300 × 1200	42328	5530	6876	3460	**2637**	4155	—	208852	276380	181817	193987	160401	6865
1300 × 1300	94325	8982	22279	**2521**	5294	5304	244973	129848	225067	165818	292777	197777	15278
1300 × 1400	30452	133087	7374	**1779**	11096	3707	259953	244078	—	256791	—	242488	11190
1300 × 1500	177355	25697	18810	4061	234235	**1998**	258119	213104	259693	172842	274986	222485	24380
1500 × 800	176	141	121	114	140	129	100	**85**	100	**85**	100	101	926
1500 × 900	25964	56674	1619	**1212**	3382	1876	141359	—	206751	95370	167671	100623	5215
1500 × 1000	91242	235192	2888	1803	5121	**1725**	—	118777	223241	93759	173623	187153	2952
1500 × 1100	—	18023	4977	7517	**3925**	4446	219596	187570	205041	179414	183862	156035	4875
1500 × 1200	139993	68562	7001	**1242**	7431	1571	251651	148664	231829	70431	254111	189705	6232
1500 × 1300	291712	—	26724	4895	20263	**4325**	—	225509	—	232735	—	255728	33785
1500 × 1400	—	6264	35073	3139	169215	**2675**	—	184555	271054	107886	—	180857	21910
1500 × 1500	—	17824	37722	2703	121472	**2528**	257242	222360	—	167762	—	251033	8699

reals, but even the baseline encoding with relaxations resulted in a timeout for every setup other than the ones for which no feasible configuration exists. We experimented also with OR-Tools on an encoding obtained considering fixed-point arithmetic over 64 bit integers, but to no avail. We did not try the fixed-point encoding on other CP tools as they do not support 64 bit precision which is the least one required to avoid overflowing the calculations. In Table 6.4 we collect the results of the comparison between z3 and OptiMathSat, adding the runtime of the heuristic search performed by LIFTCREATE for reference. We focus on the implication-based encoding given the results with the baseline encoding. In the table, columns labeled "**I**" report runtimes on the integer-based versions, columns labeled "**I+R**" report runtimes on relaxed versions, and columns labeled "**I+R+F**" report runtimes on versions where look-up tables are represented as nested *if-then-else* functions rather than straight implications. The columns "**SO**" and "**MO**" report the results of single-objective and multi-objective optimization, respectively. The choice of the weights detailed in the setup reflects that the first three components of the cost function have the same priorities. Different weights could be chosen according to the user's preferences, and we know — from other experiments that we do not show here to save space — that different choices do not impact on performances.

Considering the results in Table 6.4, we see that the integer-based version of the full encoding is the least appealing option: while z3 performs slightly better than OptiMathSat on this version, other solutions yield faster runtimes. In particular, relaxing the encoding has a substantial impact both on z3 and OptiMathSat: solving time decreases by orders of magnitude in some cases with respect to the integer-based encoding. Finally, considering the addition of native SMT-LIB functions we see that the results are mixed, i.e., it is not so clear that choosing them improves the solving time. Noticeably, while OptiMathSat remains slower than z3, it never exceeds the time limit on this encoding. As for single vs. multi-objective encoding, we can see that the multi-objective approach performs better than the single-objective one. In spite of some exceptions, multi-objective optimization — specifically, with z3, relaxed encodings and native SMT-LIB functions — seems to be the winning option overall. When it comes to comparing the heuristic engine of LIFTCREATE with the best results of the constraint-based approach, we should take into account that the former deals with the *complete* design cycle and not just with some subtasks. Given this initial bias, that in some cases the heuristic engine outperforms most constraint-based solutions, but it is overall slower than the best ones, it is fair to say that OMT solvers with relaxed encodings and multi-objective optimization provide a feasible replacement to heuristic search in the design subtasks that we considered here.

Part II

Verification of Neural Networks

Chapter 7

Abstraction algorithms

In this chapter we show the algorithms and definitions that compose our abstraction model for the verification of neural networks by means of reachability analysis and robustness certification. We give the general definitions for abstracting domains, and afterwards we focus on how to propagate this abstraction throughout the activation layers.

7.1 Basic abstraction definitions

To enable algorithmic verification of neural networks, we consider the abstract domain $\langle \mathbb{R}^n \rangle \subset 2^{\mathbb{R}^n}$ of polytopes defined in $\mathbb{R}^n$ to abstract (families of) bounded sets into (families of) polytopes. We provide corresponding abstractions for affine and functional layers to perform abstract computations and we prove that their composition provides a consistent overapproximation of concrete networks.

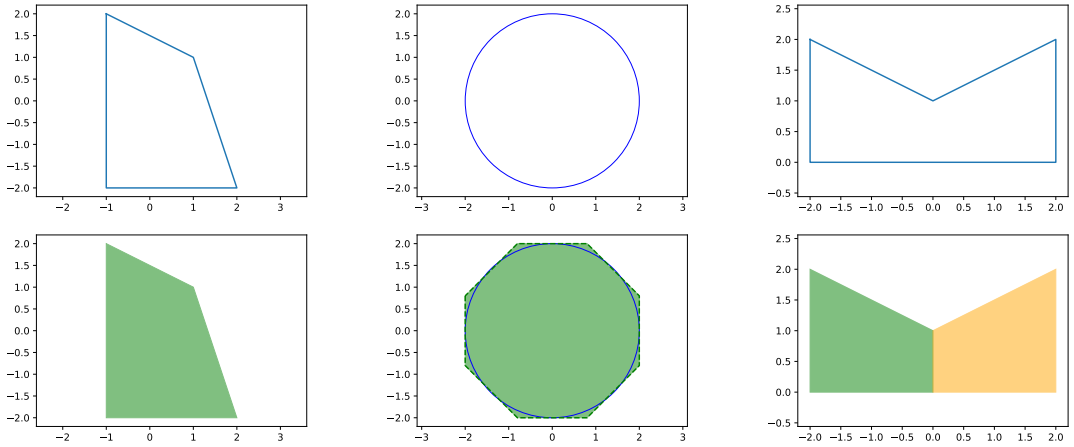
Definition 1 *(Abstraction)* Given a bounded set $X \subset \mathbb{R}^n$, an abstraction is defined as a function $\alpha : 2^{\mathbb{R}^n} \to \langle \mathbb{R}^n \rangle$ that maps X to a polytope P such that $\mathscr{C}(X) \subseteq P$.

Intutively, the function α maps a bounded set X to a corresponding polytope in the abstract space such that the polytope always contains the convex hull of X. Depending on X, the enclosing polytope may not be unique — see Figure 7.1 for different examples. However, given the convex hull of any bounded set, it is always possible to find an enclosing polytope. As shown in Zheng (2019), one could always start with an axis-aligned regular n simplex consisting of $n+1$ facets — e.g., the triangle in $\mathbb{R}^2$ and the tethraedron in $\mathbb{R}^3$ — and then refine the abstraction as needed by adding facets, i.e., adding half-spaces to make the abstraction more precise.

Definition 2 *(Concretization)* Given a polytope $P \in \langle \mathbb{R}^n \rangle$ a concretization is a function $\gamma : \langle \mathbb{R}^n \rangle \to 2^{\mathbb{R}^n}$ that maps P to the set of points cointained in it, i.e., $\gamma(P) = \{x \in \mathbb{R}^n \mid x \in P\}$.

Intutively, the function γ simply maps a polytope P to the corresponding (convex and compact) set in $\mathbb{R}^n$ comprising all the points contained in the polytope. As opposed to abstraction, the result of concretization is uniquely determined. We extend abstraction and concretization to finite families of sets and polytopes, respectively, as follows. Given a family of p bounded sets $\Pi = \{X_1, \ldots, X_p\}$, the abstraction of Π is a set of polytopes $\Sigma = \{P_1, \ldots, P_s\}$ such that $\alpha(X_i) \subseteq \bigcup_{i=1}^{s} P_i$ for all $i \in [1, p]$; when no ambiguity arises, we abuse notation and write $\alpha(\Pi)$ to denote the abstraction corresponding to the family Π. Given a family of s polytopes $\Sigma = \{P_1, \ldots, P_s\}$, the concretization of Σ is the union of the

Figure 7.1 Three possible abstractions of a set: the first row depicts the bounded set X, and the second the enclosing polytope P. Starting from the left, the first set is a convex set whose polytope matches perfectly. The second is not linear, and it is approximated with an octagon. The third is linear but non convex, therefore is split into two convex polytopes.



concretizations of its elements, i.e., $\bigcup_{i=1}^{s} \gamma(P_i)$; also in this case, we abuse notation and write $\gamma(\Sigma)$ to denote the concretization of a family of polytopes Σ.

Given our choice of abstract domain and a concrete network $\nu : I \to O$ with $I \subset \mathbb{R}^n$ and $O \subset \mathbb{R}^m$, we need to show how to obtain an *abstract neural network* $\tilde{\nu} : \langle I \rangle \to \langle O \rangle$ that provides a sound overapproximation of ν. To frame this concept, we introduce the notion of consistent abstraction.

Definition 3 *(Consistent abstraction)* Given a mapping $\nu : \mathbb{R}^n \to \mathbb{R}^m$, a mapping $\tilde{\nu} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^m \rangle$, abstraction function $\alpha : 2^{\mathbb{R}^n} \to \langle \mathbb{R}^m \rangle$ and concretization function $\gamma : \langle \mathbb{R}^m \rangle \to 2^{\mathbb{R}^m}$, the mapping $\tilde{\nu}$ is a consistent abstraction of ν over a set of inputs $X \subseteq I$ exactly when

$$\{\nu(x) \mid x \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X))) \tag{7.1}$$

The notion of consistent abstraction can be readily extended to families of sets as follows. The mapping $\tilde{\nu}$ is a consistent abstraction of ν over a family of sets of inputs $X_1 \dots X_p$ exactly when

$$\{\nu(x) \mid x \in \cup_{i=1}^{p} X_i\} \subseteq \gamma(\tilde{\nu}(\alpha(X_1, \dots, X_p))) \tag{7.2}$$

where we abuse notation and denote with $\tilde{\nu}(\cdot)$ the family $\{\tilde{\nu}(P_1), \dots, \tilde{\nu}(P_s)\}$ with $\{P_1, \dots, P_s\} = \alpha(X_1, \dots X_p)$

To represent polytopes and define the computations performed by abstract layers we resort to a specific subclass of *generalized star sets*, introduced in Bak and Duggirala (2017) and defined as follows — the notation is adapted from Tran et al. (2019).

Definition 4 *(Generalized star set)* Given a *basis matrix* $V \in \mathbb{R}^{n\times m}$ obtained arranging a set of *m basis vectors* $\{v_1, \dots v_m\}$ in columns , a point $c \in \mathbb{R}^n$ called *center* and a *predicate* $R : \mathbb{R}^m \to \{\top, \bot\}$, a generalized star set is a tuple $\Theta = (c, V, R)$. The set of points represented by the generalized star set is given by

$$[\![\Theta]\!] \equiv \{z \in \mathbb{R}^n \mid z = Vx + c \text{ such that } R(x_1, \dots, x_m) = \top\} \tag{7.3}$$

In the following we denote $[\![\Theta]\!]$ also as Θ. Depending on the choice of R, generalized star sets can represent different kinds of sets, but we consider only those such that $R(x) := Cx \leq d$, where $C \in \mathbb{R}^{p\times m}$ and $d \in \mathbb{R}^p$ for $p \geq 1$, i.e., R is a conjunction of p linear constraints as in Tran et al. (2019); we further require that the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded.

Proposition 1 *Given a generalized star set $\Theta = (c, V, R)$ such that $R(x) := Cx \leq d$ with $C \in \mathbb{R}^{p\times m}$ and $d \in \mathbb{R}^p$, if the set $Y = \{y \in \mathbb{R}^m \mid Cy \leq d\}$ is bounded, then the set of points represented by Θ is a polytope in $\mathbb{R}^n$, i.e., $\Theta \in \langle \mathbb{R}^n \rangle$.*

The proof of proposition (1) is straightforward, since the set Y is a polytope in $\mathbb{R}^m$, the mapping $Vx + c$ is an affine mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$ and affine mappings of polytopes are still polytopes. From Tran et al. (2019) we know that polytopes can be represented as generalized star sets, and thus our restricted form of star sets provides an equivalent representation of polytopes in $\mathbb{R}^n$; in the following, we refer to generalized star sets obeying our restrictions simply as *stars*.

The simplest abstract layer to obtain is the one abstracting affine transformations. As we have already mentioned, affine transformations of polytopes are still polytopes, so we just need to define how to apply an affine transformation to a star — the definition is adapted from Tran et al. (2019).

Definition 5 *(Abstract affine mapping)* Given a star set $\Theta = (c, V, R)$ and an affine mapping $f : R^n \to R^m$ with $f = Ax + b$, the abstract affine mapping $\tilde{f} : \langle R^n \rangle \to \langle R^m \rangle$ of f is defined as $\tilde{f}(\Theta) = (\hat{c}, \hat{V}, R)$ where

$$\hat{c} = Ac + b \qquad \hat{V} = AV$$

Intuitively, the center and the basis vectors of the input star Θ are affected by the transformation of f, while the predicates remain the same.

Proposition 2 *Given an affine mapping $f : \mathbb{R}^n \to \mathbb{R}^m$, the corresponding abstract mapping $\tilde{f} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^m \rangle$ provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{f(x) \mid x \in X\} \subseteq \gamma(\tilde{f}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$.*

To prove proposition (2), we observe that the set $\alpha(X)$ is any polytope P such that $P \supseteq \mathscr{C}(X)$ — equality holds only when X is already a polytope, and thus $X \equiv \mathscr{C}(X) \equiv P$. Let $\Theta_P = (c_P, V_P, R_P)$ be the star corresponding to P defined as

$$c_P = 0^n \qquad V_P = I^n \qquad R_P = C_P x + d_P \leq 0$$

where 0^n is the n-dimensional zero vector, and I^n is the $n \times n$ identity matrix — the columns of I^n correspond to the standard orthonormal basis $e_1, \ldots, e_n$ of $\mathbb{R}^n$, i.e., $\|e_i\| = 1$ and $e_i \cdot e_j = 0$ for all $i \neq j$ with $i, j \in [1, n]$; the matrix $C_P \in \mathbb{R}^{q \times n}$ and the vector $d_P \in \mathbb{R}^q$ collect the parameters defining q half-spaces whose intersection corresponds to P. Given our choice of c and V, it is thus obvious that $\Theta_P \equiv P$. Recall that $f = Ax + b$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$; from definition (5) we have that $\tilde{f}(\Theta_P) = \hat{\Theta}_P$ with $\hat{\Theta}_P = (\hat{c}_P, \hat{V}_P, R_P)$ and

$$\hat{c}_P = A0^n + b = b \qquad \hat{V}_P = AI^n = A$$

The concretization of $\hat{\Theta}_P$ is just the set of points contained in $\hat{\Theta}_P$ defined as

$$\gamma(\hat{\Theta}_P) = \{z \in \mathbb{R}^m \mid z = Ax + b \text{ such that } C_p x \leq d_P\} \tag{7.4}$$

Now it remains to show that $\{f(x) \mid x \in X\} \subseteq \gamma(\hat{\Theta}_P)$. This follows from the fact that, for a generic $y \in \{f(x) \mid x \in X\}$ there must exists $x \in X$ such that $y = Ax + b$; since x satisfies $C_p x \leq d_P$ by construction of P, it is also the case that $y \in \gamma(\hat{\Theta}_P)$ by definition (7.4).

7.2 ReLU abstraction algorithms

Algorithm 2 Guidotti et al. (2021) defines the abstract mapping of a functional layer with n ReLU activation functions and adapts the methodology proposed in Tran et al. (2019). The function COMPUTE_LAYER takes as input an indexed list of N stars $\Theta_1, \ldots, \Theta_N$ and an indexed list of n positive integers called *refinement levels*. For each neuron, the refinement level tunes the grain of the abstraction: level 0 corresponds to the coarsest abstraction that we consider — the greater the level, the finer the abstraction grain. In the case of ReLUs, all non-zero levels map to the same (precise) refinement, i.e., a piecewise affine mapping. The

Algorithm 2 Abstraction of the ReLU activation function.

1: **function** COMPUTE_LAYER(*input* = $[\Theta_1,\ldots,\Theta_N]$, *refine* = $[r_1,\ldots,r_n]$)
2: *output* = [ ]
3: **for** $i = 1 : N$ **do**
4: *stars* = $[\Theta_i]$
5: **for** $j = 1 : n$ **do** *stars* = COMPUTE_RELU(*stars*, j, *refine*$[j]$, n)
6: APPEND(*output*, *stars*)
7: **return** *output*

8: **function** COMPUTE_RELU(*input* = $[\Gamma_1,\ldots,\Gamma_M]$, j, *level*, n)
9: *output* = [ ]
10: **for** $k = 1 : M$ **do**
11: (lb_j, ub_j) = GET_BOUNDS(*input*$[k]$, j)
12: $M = [e_1 \ldots e_{j-1}\ 0\ e_{j+1} \ldots e_n]$
13: **if** $lb_j \geq 0$ **then** S = *input*$[k]$
14: **else if** $ub_j \leq 0$ **then** $S = M$ * *input*$[k]$
15: **else**
16: **if** *level* > 0 **then**
17: Θ_{low} = *input*$[k] \wedge z[j] < 0$; Θ_{upp} = *input*$[k] \wedge z[j] \geq 0$
18: $S = [M * \Theta_{low}, \Theta_{upp}]$
19: **else**
20: $(c, V, Cx \leq d)$ = *input*$[j]$
21: $C_1 = [0\ 0\ \ldots\ -1] \in \mathbb{R}^{1\times m+1}$, $d_1 = 0$
22: $C_2 = [V[j,:]\ -1] \in \mathbb{R}^{1\times m+1}$, $d_2 = -c_k[j]$
23: $C_3 = [\frac{-ub_j}{ub_j - lb_j} \cdot V[j,:]\ -1] \in \mathbb{R}^{1\times m+1}$, $d_3 = \frac{ub_j}{ub_j - lb_j}(c[j] - lb_j)$
24: $C_0 = [C\ 0^{m\times 1}]$, $d_0 = d$
25: $\hat{C} = [C_0;\ C_1;\ C_2;\ C_3]$, $\hat{d} = [d_0;\ d_1;\ d_2;\ d_3]$
26: $\hat{V} = MV$, $\hat{V} = [\hat{V}\ e_j]$
27: $S = (Mc, \hat{V}, \hat{C}\hat{x} \leq \hat{d})$
28: APPEND(*output*, S)
29: **return** *output*

output of function COMPUTE_LAYER is still an indexed list of stars, that can be obtained by independently processing the stars in the input list. For this reason, the **for** loop starting at line 3 can be parallelized to speed up actual implementations.

Given a single input star $\Theta_i \in \langle R^n \rangle$, each of the n dimensions is processed in turn by the **for** loop starting at line 5 and involing the function COMPUTE_RELU. Notice that the stars obtained processing the j-th dimension are feeded again to COMPUTE_RELU in order to process the $j+1$-th dimension. For each star given as input, the function COMPUTE_RELU first computes the lower and upper bounds of the star along the j-th dimension by solving two linear-programming problems — function GET_BOUNDS at line 11. Independently from

the abstraction level, if $lb_j \geq 0$ then the ReLU acts as an identity function (line 13), whereas if $ub_j \leq 0$ then the j-th dimension is zeroed (line 14). The $*$ operator takes a matrix M, a star $\Gamma = (c, V, R)$ and returns the star (Mc, MV, R). In this case, M is composed of the standard orthonormal basis in $\mathbb{R}^n$ arranged in columns, with the exception of the j-th dimension which is zeroed.

7.2.1 Exact abstract propagation

When $lb_j < 0$ and $ub_j > 0$ we consider the refinement level. For any non-zero level, the input star is "split" into two new stars, one considering all the points $z < 0$ (Θ_{low}) and the other considering points $z \geq 0$ (Θ_{upp}) along dimension j. Both Θ_{low} and Θ_{upp} are obtained by adding to the input star *input[k]* the appropriate constraints. If the analysis at lines 17–18 is applied throughout the network, and the input abstraction is precise, then the abstract output range will also be precise, i.e., it will coincide with the concrete one: we call complete the analysis of NEVER2 in this case. The number of resulting stars is worst-case exponential, therefore the complete analysis may result computationally infeasible.

Proposition 3 *Given a ReLU mapping $f : \mathbb{R}^n \to \mathbb{R}^n$, the corresponding abstract mapping $\tilde{f} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^n \rangle$ defined in Algorithm 2 provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{f(x) \mid x \in X\} \subseteq \gamma(\tilde{f}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$.*

Proposition 4 *Given a concrete network $\nu : \mathbb{R}^n \to \mathbb{R}^m$ comprised of a finite number p of layers $f_1 : \mathbb{R}^n \to \mathbb{R}^{n_1}, \ldots, f_p : \mathbb{R}^{n_{p-1}} \to \mathbb{R}^m$ such that each f_i is either an affine or functional layer implementing ReLUs, the corresponding abstract network $\tilde{\nu} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^m \rangle$ comprised of the corresponding abstract layers $\tilde{f}_1 : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^{n_1} \rangle, \ldots, \tilde{f}_p : \langle \mathbb{R}^{n_{p-1}} \rangle \to \langle \mathbb{R}^m \rangle$ provides a consistent abstraction over any bounded set $X \subset \mathbb{R}^n$, i.e., $\{\nu(x) \mid x \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X)))$ for all $X \subset \mathbb{R}^n$.*

Proposition 4 enforces that we can prove the (local) robustness of a neural network by propagating the abstraction of an input set representing the l_∞ ball around a given input with a small perturbation ε and check whether the output set is large enough to cause a misclassification.

Proposition 5 *The intersection of a star $\Theta = (c, V, R)$ and a half-space $\mathscr{H} = \{z | Hz \leq g\}$ is another star with the following characteristics: $\overline{\Theta} = \Theta \cap \mathscr{H} = (\overline{c}, \overline{V}, \overline{R})$ with $\overline{c} = c$, $\overline{V} = V$, $\overline{R} = R \wedge R'$ and $R'(x) = (HV)x \leq g - Hc$*

The proof of the proposition is straightforward since it is analogous to adding new constraints to the predicate of the star, as done for the ReLU abstract transformer.

Proposition 6 *Let $[\Theta_1,...,\Theta_n]$ be a star set obtained by applying Algorithm 2 to a network of interest ν and an input star set corresponding to the input component of the property of interest P. Moreover let $\hat{\mathscr{H}}$ be an half-space corresponding to the unsafe zone as defined by the property of interest. If $\overline{\Theta}_i = \Theta_i \wedge \hat{\mathscr{H}} = \emptyset$ for $i = 1,...,n$ then the neural network ν satisfy the property P.*

Proposition 7 *If in Algorithm 2 the stars were always refined for all neurons it is possible to compute the complete counter input set (i.e., the set containing all possible inputs that make the neural network unsafe) as $\mathscr{C}_\Theta = \bigcup_i (c,V,\overline{R}_i)$ where $\overline{R}_i$ are the predicates of the stars obtained by the intersection between the unsafe zone and the output star set, whereas c and V are the center and basis matrix of the input star.*

Proof 1 *If the complete version of Algorithm 2 is used then all the stars in the computation process are defined on the same predicate variables $\boldsymbol{x} = [x_1,...,x_m]$ which do not change during the computations since only the number of constraints on $\boldsymbol{x}$ is changed by the abstract transformers. As consequence the $\overline{R}_i$ contain values of $\boldsymbol{x}$ that make the network unsafe, moreover it also contains all the constraints of the base predicate R of the input star. Therefore the complete counter input set containing all possible inputs that make the neural network unsafe is $\mathscr{C}_\Theta = \bigcup_i (c,V,\overline{R}_i)$, $\overline{R}_i \neq \emptyset$.*

7.2.2 Over-approximate abstract propagation

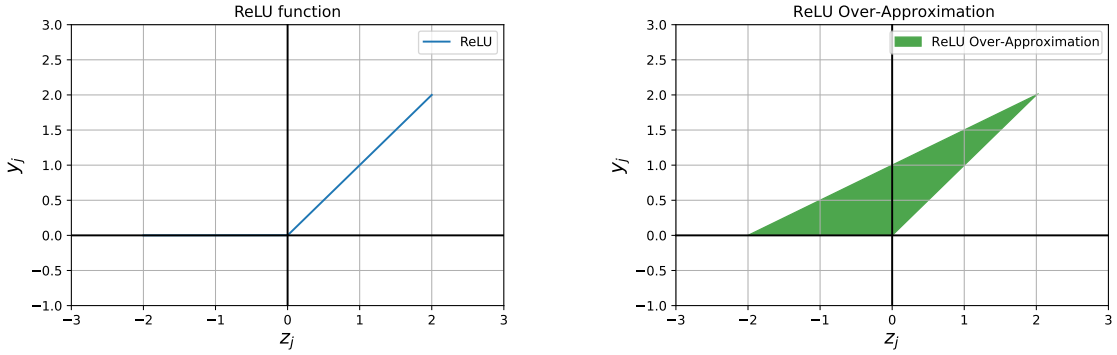
If the refinement level is 0, then the ReLU is abstracted using the over-approximation proposed in Tran et al. (2019) and depicted in Figure 7.2. This approach is much less conservative than others, i.e., based on zonotopes or abstract domains, and provides a tighter abstraction.

As can be seen in Figure 7.2, three constraints are needed to construct the over-approximation:

$$
\begin{aligned}
& y_j \geq 0 \\
& y_j \geq z_j \\
& y_j \leq ub_j \frac{z_j - lb_j}{ub_j - lb_j}
\end{aligned}
$$

Such constraints must be added to the predicate matrix of the star, therefore we define an auxiliarly variable x_{m+1} and we modify the basis matrix so that $y_j = x_{m+1}$ (line 26 in

Figure 7.2 Graphical representation of the ReLU function (left) and the over-approximation considering a single variable (right) with $lb_j = -2$ and $ub_j = 2$.



Algorithm 2). By doing so we make it possible to express our constraints only in terms of the predicate variables. We remember that $z_j = V_j\mathbf{x} + c_j$, substituting it in the constraints we obtain:

$$
\begin{aligned}
x_{m+1} &\geq 0 \\
x_{m+1} &\geq V_j\mathbf{x} + c_j \\
x_{m+1} &\leq ub_j \cdot \frac{V_j\mathbf{x} + c_j - lb_j}{ub_j - lb_j}
\end{aligned}
$$

If we reorder these constraints we can bring them in the format $C\mathbf{x} \leq \mathbf{d}$:

$$
\begin{aligned}
-x_{m+1} &\leq 0 \\
V_j\mathbf{x} - x_{m+1} &\leq -c_j \\
-\frac{ub_j}{ub_j - lb_j}V_j\mathbf{x} + x_{m+1} &\leq \frac{ub_j}{ub_j - lb_j}(c_j - lb_j)
\end{aligned}
$$

From these constraints it is straightforward to identify the corresponding matrices in lines 21 to 23 of the algorithm.

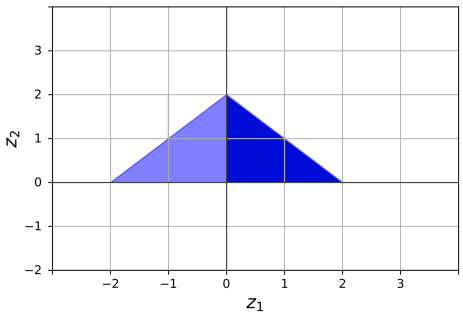
If this analysis is carried out throughout the network, then the output star will be a (sound) over-approximation of the concrete output range: we call *over-approximate* the analysis of NEVER2 in this case. The number of star remains the same throughout the analysis, but at the cost of a new predicate variable for each neuron which, in turn, increases the complexity of the linear program required by GET_BOUNDS.

7.2.3 Mixed abstract propagation

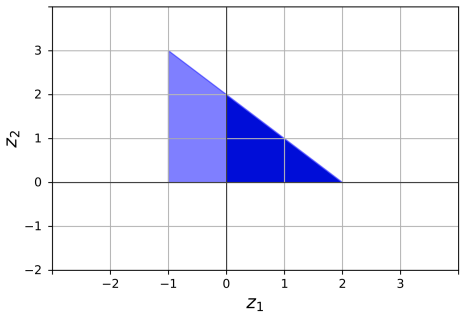
In Guidotti et al. (2021) it is proposed a new approach that adopts different levels of abstraction during the analysis: since each neuron features its own refinement level, algorithm 2 controls the abstraction down to the single neuron. This setting strikes a trade-off between complete and over-approximate settings. In order to reduce as much as possible the approximation error, we rank the neurons in each layer based on the area of the over-approximation triangle depicted in Figure 7.2: intuitively, the neuron with the widest bounds introduces a broader triangle and, by design, a bigger approximation.

We concretize the star along that neuron and propagate the approximate method along the others, such that each layer results in at most a single split. This reduces the computational cost significantly, as the growth becomes quadratic in the number of layers and the complexity increase by the approximation is contained. We call *mixed* the analysis of NEVER2 in this case.

Figure 7.3 ReLU split subsumption example along axis z_1 (the actual lower star is collapsed to a line)



(a) In this example the lower star can be subsumed by the upper one



(b) In this example the lower star cannot be subsumed by the upper one

7.3 Improving abstract propagation

The abstraction procedure detailed in Section 7.1 allows to control the number of stars produced during the layer propagation. Nevertheless, we can note that the lower star in the ReLU split could be completely subsumed by the upper one depending on the bounds along the other variables.

Figure 7.3 exemplifies this statement: in case 7.3a the lower star is negligible when projected to the z_2 axis while in case 7.3b the projection adds information which is not present in the upper star. More formally, we can say that if the lower star bounds along the dimension z_2 are lesser than the upper star ones, then the upper star subsumes the lower on dimension z_2. We can generalize by stating that for the dimension, i.e., neuron i we can ignore the lower star if and only if for all other dimensions, i.e., neurons $j = 1, ..., n$, $i \neq j$:

$$ub^j_{upp} \leq ub^j_{low} \wedge lb^j_{upp} \geq lb^j_{low}$$

The procedure is sound because the star set we obtain by the COMPUTE_RELU function with the exact method in the same layer is guaranteed to contain stars with the same number of dimensions such that the comparison is possible. Furthermore, given that ReLU does not perform affine transformations, all stars share the same center and the dimension in the basis matrix which is zeroed corresponds to the projection of the lower star on that dimension.

Algorithm 3 details the procedure for checking the elimination of the lower star. In order to contain the number of LPs to solve, since each dimension is processed subsequently, we start checking the dimensions following the one alongside which the ReLU split occurred:

Algorithm 3 Star elimination algorithm

```
1:  function GET_UNIQUE(lower, upper, v)
2:      lower.lb_v = lower.ub_v = 0
3:      upper.lb_v = 0
4:      dim_list = ORDER(tot_vars, v)
5:      for j in dim_list do
6:          lb_low_j, ub_low_j = GET_BOUNDS(lower, j)
7:          lb_upp_j, ub_upp_j = GET_BOUNDS(upper, j)
8:          if ub_upp > ub_low or lb_upp < lb_low then
9:              return [lower, upper]
10:     return [upper]

11: function ORDER(num_vars, j)
12:     output = [ ]
13:     for i = j+1 : num_vars do
14:         APPEND(output, i)
15:     for i = 0 : j−1 do
16:         APPEND(output, i)
17:     return output

18: function COMPUTE_RELU(input = [Γ_1, ..., Γ_M], j, level, n)
        ...
19:     if level > 0 then
20:         Θ_low = input[k] ∧ z[j] < 0;  Θ_upp = input[k] ∧ z[j] ≥ 0
21:         S = GET_UNIQUE(M * Θ_low, Θ_upp, j)
        ...
```

in this way, even if the check fails, the LP does not add extra computational time as it is used for the next neuron. This is obtained by function ORDER in line 4. After ordering the dimensions, we perform the subsumption check for each one of them; whenever this check fails, both the lower and the upper star are returned without spending further time checking other dimensions (line 9). Only if the check is successful for every dimension, the function returns the upper star only (line 10). This modification is highlighted in the fragment of COMPUTE_RELU in lines $18-22$. The cost for the extra LPs paid when the upper star subsumes the lower is balanced by the reduction of the number of stars that are propagated, whereas if the subsumption check fails soon enough, no extra LPs are computed since the bounds are used in the next neuron computation. The worst case scenario is a check that takes almost all the dimensions to finally fail, which leads eventually to a major overhead.

7.4 Counter-example Guided Abstraction Refinement

Our algorithm can be used to compute the complete or over-approximate reachable set of the neural network of interest. Once the reachable set has been computed, the property of interest can be verified by computing the intersection between the negation of such property and the reachable set (which we call *reachable counter set*). If such intersection is the empty set, then the network is compliant with the property of interest; otherwise, if the reachable set is complete, we have shown that the network is unsafe. However, if the reachable set is over-approximated, the concrete network may satisfy the property, and the over-approximation may be too coarse. In both cases in which the reachable counter set is not the empty set, we are interested in extracting concrete input points corresponding to the output contained in the reachable counter set. In particular, when we have a complete counter reachable set we can leverage the following theorem, adapted from Tran (2020):

Theorem 1 *Let ν be a feed-forward neural network, $\Theta = (c,V,R)$ be a star input set, $\nu(\Theta) = \bigcup_{i=1}^{k} \Theta_i$, $\Theta_i = (c_i,V_i,R_i)$ be the reachable set of the neural network and* S *be a safety specification. Denote $\overline{\Theta}_i = \Theta_i \cap \neg \mathbf{S} = (c_i,V_i,\overline{P}_i)$, $i = 1,...,k$. The neural network is safe if and only if $\overline{P}_i = 0$ for all i. If the neural network violates its safety property then the complete counter input set containing all possible inputs in the input set that lead the neural network to unsafe states is $\mathbf{C} = \bigcup_{i=1}^{k}(c,V,\overline{P}_i), \overline{P}_i \neq 0$.*

For the proof of Theorem 1 we refer to Tran (2020). Using Theorem 1 we can easily compute the complete counter input set, so the problem of extracting concrete input points becomes the problem of extracting points from a star-set which in itself can be considered as extracting points from a single star. To do this, we consider the problem of extracting points from the predicate of the star, which, under our pre-conditions, is always a polytope. We will then apply to the points of the predicate (α) the affine transformation $x = c + V\alpha$ to obtain a corresponding point of the star of interest. To extract the point from the polytope defined by the predicate, we leverage the hit and run sampler Smith (1996). It should be noted that while the hit and run algorithm produces an approximation of a uniform distribution for the α of the predicate, the application of the affine transformation needed for the transformation to the point of the star skews such distribution. A possible solution to this issue is to transform the predicate to its V-representation, apply the affine transformation directly to the polytope, return to the H-representation and apply the hit and run sampler. However, for our aims, the skew of the distribution is not that relevant. Therefore, at least at this time, we do not need to transform between the two representations, which is computationally expensive. The

problem is different when we are working with the over-approximate reachable counter set: in this case, we do not have a way to compute the counter input set since the addition of the new variables needed for the over-approximation to the predicate of the star invalidates Theorem 1. Therefore an alternative solution is needed to compute inputs that allegedly are not compliant with the property of interest. We define the *abstract counter output set* (ACOS) as the intersection between the abstract reachable set and the negation of the property S. Our algorithm extracts a point from the ACOS using hit and run sampling and then searches for the corresponding input point. Formally the search problem of the corresponding input point can be defined as:

Definition 6 *Given a reference output point $\hat{y}$, a starting input point x and a feed forward neural network ν we can define the search problem for the point $\hat{x}$ which satisfies $\nu(\hat{x}) = \hat{y}$ as the following minimization problem:*

$$\hat{x} = \min_{x} ||\hat{y} - \nu(x)||_2$$

However, the non-convexity and non-linearity of the function make the minimization problem not easily solvable: the non-convexity and the presence of local minima make it extremely difficult to apply gradient descent. Consequently, we developed a simple search-by-sampling algorithm which, given a starting point in the input space, generates a “cloud" of points using a normal distribution with the starting point as center and a given variance. Such points are then compared, and the one whose corresponding output is nearest to the desired one is selected as the center for another step of the algorithm. The search terminates when the euclidean distance between the output found and the one we are searching for is less than a given threshold or when a given number of steps is exceeded. If the algorithm finds an input point in the concrete input set and whose corresponding output is in the ACOS, we have found a concrete counter-example, and the network is proven unsafe. Otherwise, the point found is a point whose corresponding output is reasonably close to the ACOS and can be leveraged for our refinement. Once an adequate sample is found, we can use it to guide our refinement. The idea behind the refinement algorithm is to rank the approximation error for each neuron by computing the triangle areas of the approximate method — see, Section 7.2 — and enhance it with a measure of the relevance of the neurons with respect to the sample found. To compute the relevance, we leveraged the layer-wise relevance propagation algorithm Samek et al. (2016) which, while traditionally used by the explainability community for classification models, can provide an adequate relevance measure even for regression tasks. It should be noted that our implementation of the algorithm supports, at present, only fully-connected

Algorithm 4 CEGAR Algorithm.

```
1:  function CEGAR_VERIFICATION(input_set, unsafe_zone, network)
2:      ref_levels = [0, ..., 0]
3:      ACOS, areas, safe = STARSET_VER(input_set, unsafe_zone, network, ref_levels)
4:      if IS_EMPTY(ACOS) then
5:          return ACOS, areas, True

6:      output_counter = GET_SAMPLE(ACOS)
7:      input_counter = INPUT_SEARCH(network, output_counter)
8:      if input_counter ∈ input_set then
9:          return ACOS, areas, False

10:     neuron_relevances = COMPUTE_REL(input_counter, network)
11:     ref_levels = COMPUTE_REF_LEVELS(neuron_relevances, areas)
12:     return STARSET_VER(input_set, unsafe_zone, network, ref_levels)
```

layers and ReLU activation functions. For more details on layer-wise relevance propagation we refer to Montavon et al. (2019).

The refinement procedure is detailed in Algorithm 4 Demarchi and Guidotti (2022). As the first thing, it needs to apply our verification methodology in its over-approximate form (line 3) to compute the over-approximate reachable counter set and the triangle areas. If the network is proven to be safe (line 4) then the verification algorithm terminates (line 5), otherwise we can search the counter-example as shown before (line 6 and 7). If we found a concrete counter-example then the network is proven to be unsafe and the procedure terminates (line 8 and 9), otherwise we use the spurious counter-example to find the relevances of the neurons of the network (line 10). At this point, the relevances and the triangle areas can be used to evaluate the significance of each ReLU neuron of the network. Once a measure of the significance is computed for each neuron of each ReLU layer, we can choose a given number of neurons to refine for each layer (line 11), and we can change the refinement levels of Algorithm 2 as needed. Then our verification methodology is applied again using the new refinement levels (line 12). Concerning the measure of significance, we investigate on *Product significance* (PS) which computes, for each neuron, the value of the multiplication between its relevance and the area of the triangle abstraction, and *mixed-R* (mR), which uses the relevances as coefficients for the ranking used in the standard mixed methodology.

Chapter 8

NeVerTools: CoCoNet and NeVer2

In this chapter we describe our tools CoCoNet and NeVer2 that are part of the *NeVerTools* suite. The two tools are complementary and serve two purposes: CoCoNet aims to bridge the gap between the representation and the conversion of neural networks in the verification community; on the other hand, NeVer2 is our graphical user interface (GUI) for learning and verification. Both CoCoNet and NeVer2 are written in Python and rely on two components: the pyNeVer API Guidotti et al. (2021) which contains the actual methods for the training and verification and the PyQt5 API Summerfield (2007) which allows to design GUIs for a desktop environment.

8.1 CoCoNet

The verification community has stabilized in the recent years and started in 2020 the International Verification of Neural Networks Competition (VNN-COMP) Müller et al. (2022) that gathers verification benchmarks for evaluating the performances of verification tools. In order to even the workflow and process of verification tools, the VNN-COMP relies on the VNN-LIB Guidotti et al. (2023) standard, which consists of the ONNX Bai et al. (2023) file format for the neural networks and the SMT-LIB Barret et al. (2017) for the specification of the property. Since there exist many different formats for the representation of neural networks, we developed CoCoNet [1] for allowing researchers to convert their networks to ONNX. It is also possible to build a network from scratch visually in the environment.

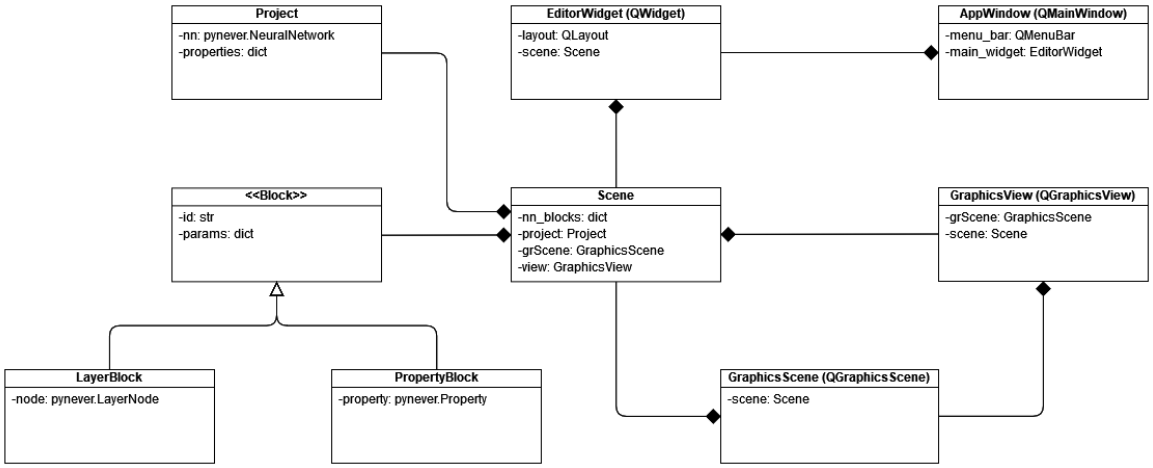
8.1.1 Software architecture

The architecture of CoCoNet is detailed in Figure 8.1. Relying on PyQt5's model for building GUIs, we use the classes `GraphicsScene` and `GraphicsView` to control the logical model and the rendering, respectively. In particular, we separate the building elements (blocks and edges) in the class `Scene` from the methods and utilities for managing the exchange with the view in `GraphicsScene`. The class `Project` serves as a controller for reflecting the actions taken in the graphical environment to the neural network that is built within pyNeVer. The main window of the application is displayed by the class `EditorWidget` which is associated with the `CoCoNetWindow` class containing the logic for displaying the `GraphicsView` and all the menus and layouts involved.

In order to comply with the VNN-LIB standard we have two kinds of available blocks: the `LayerBlock` that represent the layers of a neural network, and the `PropertyBlock` that

[1] https://github.com/NeVerTools/CoCoNet

Figure 8.1 UML Class Diagram representing the main software components of CoCoNet. Using the PyQt API we leverage the `QGraphicsView` and `QGraphicsScene` interfaces to build a workspace in the `QMainWindow`. On the other hand, the class `Scene` serves as a controller for the creation and display of graphics blocks and as an interface to the PyNeVer components.



represent a property to link to the input or the output of the network. Each `LayerBlock` is initialized with a `LayerNode` object from PyNeVer, i.e., represents a layer of the neural network displaying all the fields and allowing the modification of some parameters. `PropertyBlock` objects are initialized with a PyNeVer property, which allows to define a SMT-based rule on the input and/or the output.

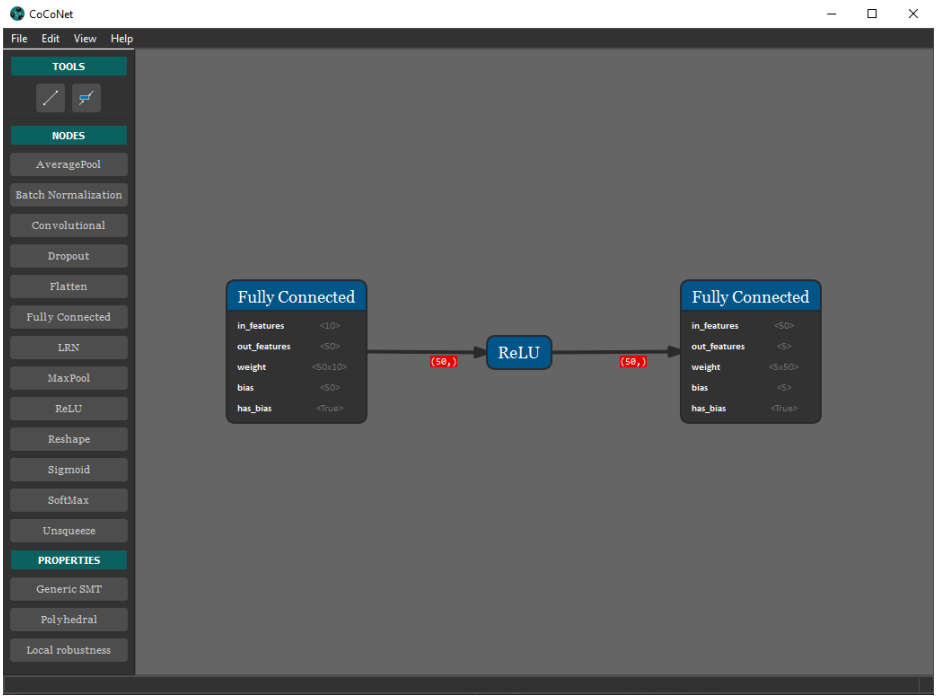
8.1.2 Application interface

Figure 8.2 is a screenshot of CoCoNet's main window. The view contains a simple network with a single Fully Connected layer with a ReLU activation function. The available layers from PyNeVer are displayed in the left toolbar, where the last three entries are different versions of a property. The displayed nodes show the information related to the corresponding node: while the ReLU layer has no parameters, the Fully Connected layer shows the inputs, the outputs and the dimension of the weight and bias matrices. When a network is loaded or created in the view, it is possible to add properties or save it in the VNN-LIB format creating one ONNX file for the network and a SMT-LIB file for the property.

It is possible to connect one or more properties to the neural network displayed, in the left toolbar we have three alternatives:

- *Generic SMT* - a simple text dialog where it is possible to directly write a property in plain SMT-LIB language

Figure 8.2 Screenshot of CoCoNet's GUI. The network is displayed in the Graphics Scene, and there is a toolbar with the available blocks divided in nodes and properties on the left.

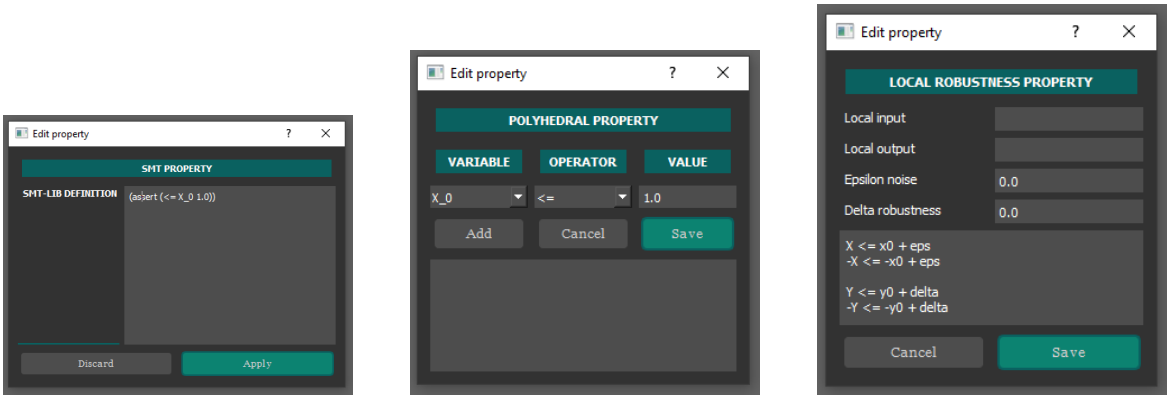


- *Polyhedral* - a property that allows to set an upper or lower bound to each variable the property is connected to
- *Local robustness* - a property that allows to specify a pair of input and output samples with an ε-δ perturbation

In Figure 8.3 we show the dialogs for the specification of the different properties. For the *Polyhedral* property the variables are constrained to the number of inputs for a *pre*-condition or the number of outputs for a *post*-condition and can be bounded with all the relational operators, i.e., $<, \leq, =, >, \geq$. The *Local robustness* property requires two samples, one for the input and one for the output, and the two ε and δ measures.

Once the network and the properties are set, it is possible to save the benchmark in the VNN-LIB format: using the *Save as...* menu, it is possible to select *VNN-LIB* as the output file format. In this way, the neural network will be saved — or converted, if opened as a different format — as an ONNX file and the properties will be stored in a separate SMT-LIB file with the same name of the network.

Figure 8.3 Screenshot of the edit dialogs for three properties available in CoCoNet. From left to right, as described in the dialog label, there is the *Generic SMT* property, the *Polyhedral* property and the *Local robustness* property.



8.2 NEVER2

Given the interest on providing a community tool for the conversion and management of neural networks we built COCONET as a stand-alone tool. Nevertheless, COCONET leverages almost every functionality of PYNEVER and lacks only the core features of learning and verification. For this reason, we built NEVER2 as a "twin" to COCONET with the extended functionalities, with the purpose of letting COCONET grow within the community and keeping NEVER2 as the interface for our verification algorithms.

A neural network created or imported in NEVER2 can be trained using a visual proxy to a PYTORCH[2]-based training procedure: every parameter for controlling the training algorithm is accessible and accounts for all kinds of datasets. It is possible to load from the user machine a custom dataset, providing the data type, the number of samples and the delimiter. Finally, the main focus of NEVER2 is the verification of a network. Altough our current capabilities do not cover all the network architectures, we provide a verification interface for fully connected neural networks with ReLU activation functions. The verification procedure requires a trained network, but within this environment it is possibile to take on the complete procedure from scratch.

8.2.1 Learning

In Figure 8.4 we show the design of the training dialog. The current abstraction of a training strategy features a single procedure which requires the neural network and a dataset instance, and updates the network displayed with new weights and biases. Currently, we have designed and implemented a single training procedure based on the **Adam** optimizer Kingma and Ba (2017). Our implementation requires a PYTORCH representation to train the network, but this is handled transparently by PYNEVER which converts the network before training.

First, it is required to load a dataset to train the network. We made available the MNIST and Fashion MNIST datasets directly, since they can be downloaded within PYTORCH without requiring extra steps. It is also possible to load a custom dataset by the user, given some extra parameters visible in Figure 8.5. The custom dataset requires to define the target index, i.e., the index for each row that distincts the inputs from the outputs, the data type and the delimiter character for the dataset entries. After the dataset selection, in the "Dataset transform" entry, it is also possible to specify some transformation functions for the input and/or the output, depending on the needs.

[2]https://pytorch.org

Figure 8.4 Screenshot of the training dialog in NEVER2. In this example it is pre-loaded for a MNIST dataset with all the default parameters of the Adam optimizer.

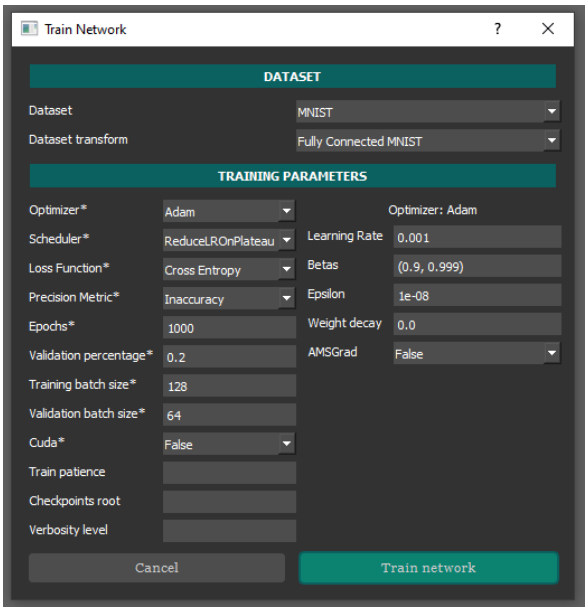
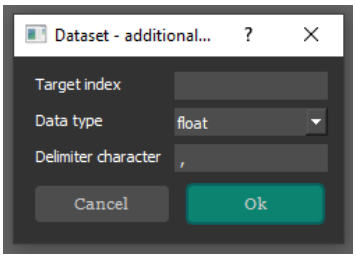
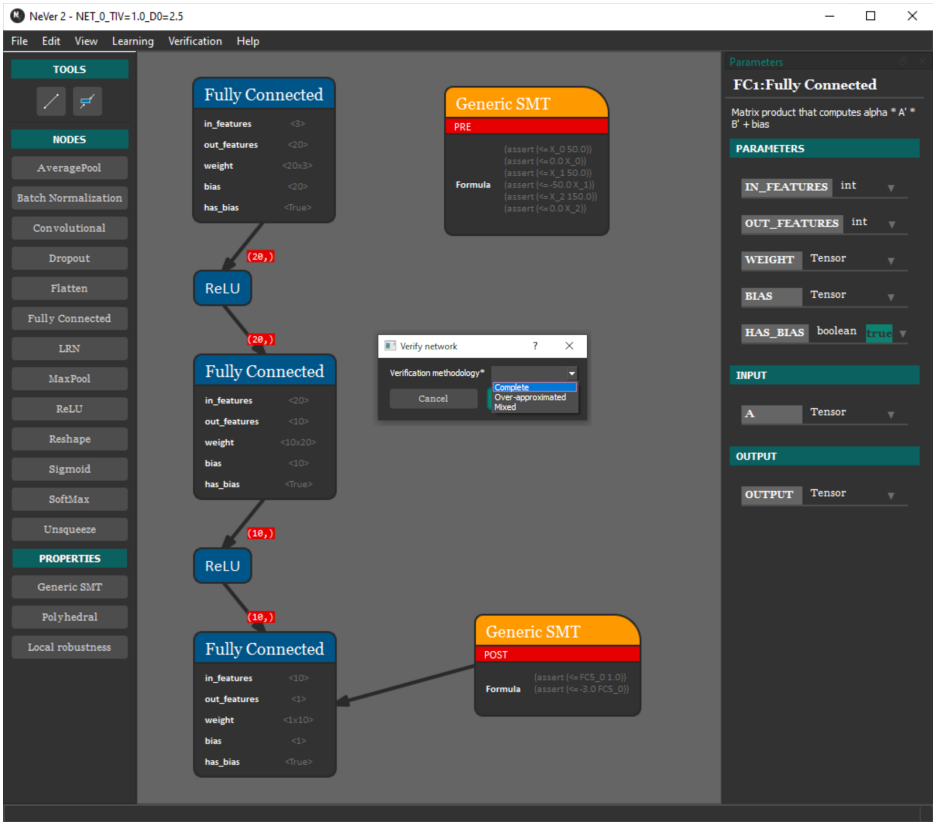


Figure 8.5 Screenshot of the dataset dialog in NEVER2. It provides default values for the data type (*float*) and delimiter character (,).



Once the dataset setup is completed, a number of training parameters is available to the user in order to tune the learning algorithm: the *Optimizer* is, for now, only **Adam** and the *Scheduler* only **ReduceLROnPlateau**. Both the optimizer and the scheduler have further parameters that are accessible in the right side of the dialog — in Figure 8.4 we see the parameters related to the Adam optimizer. We can select the *Loss function* to be either **Cross Entropy** or **MSE loss**, as well as the *Precision metric* to be either **Inaccuracy** or **MSE loss** too. Then, we can choose the number of training *Epochs*, the share of the dataset to be used as the *Validation* set and the *Training batch* and *Validation batch* sizes. Finally, it is possible to leverage the *CUDA* cores of the GPU, if the architecture supports it, to set an early stopping criterion via the *Train patience*, change the directory in which *Checkpoints* are stored and control the *Verbosity level*.

Figure 8.6 Screenshot of NEVER2 with a loaded property and the verification dialog open. Given a trained network and a property it is possible to launch the verification using one of the three algorithms provided.



8.2.2 Verification

In Figure 8.6 we show the verification dialog on a NEVER2 window. The network is already trained and there is a property attached — the orange blocks. The dialog requires to choose the verification strategy based on the different abstract propagation algorithms: *Complete* for the exact method, *Over-approximated* and *Mixed* for the approximate ones; the *Mixed* method requires to specify the number of neurons to refine per layer. The verification procedure logs the layers abstraction and returns *True* or *False* depending on whether the property is verified or not. Within PYNEVER there are two kinds of properties: `NeVerProperty` and `LocalRobustnessProperty`. `NeVerProperty` represents a generic property based on the VNN-LIB standard, meaning that is parsed by reading a SMT-LIB file and consists of the input bounds and output unsafe regions. On the other hand, `LocalRobustnessProperty` is a "pre-cooked" property encoding the search of an adversarial example corresponding to a specific data sample. PYNEVER specifies also different verification strategies, namely

`NeVerVerification`, i.e., the main contribution detailed in Guidotti et al. (2021) and Demarchi et al. (2022), and a refinement-based variation presented in Demarchi and Guidotti (2022), namely `NeVerVerificationRef`. Altough complete and ready to use, we chose not to expose all the verification interfaces in NEVER2 since they are too much experimental for the moment; should the verification benchmarks and the community benefit from this implementation, we will expose them in a future version.

Chapter 9

Experimental analysis

9.1 Case studies

Here we describe our case studies developed in order to experiment with our algorithms and to build new benchmarks for the verification community: Adaptive Cruise Control Demarchi et al. (2022) and drone control. The purpose of creating a new benchmark for the evaluation of verification algorithms is that, while the verification community has been prolific in developing novel methodologies, very few general benchmarks have been proposed, among which the most popular is still the ACAS XU benchmark Katz et al. (2017), released in 2017.

Furthermore, autonomous driving and drone control are tasks relevant for modern applications and, at the same time, the neural networks used in this kind of control are usually small enough for the existing verification methodologies to be successfully applied.

9.1.1 Adaptive Cruise Control

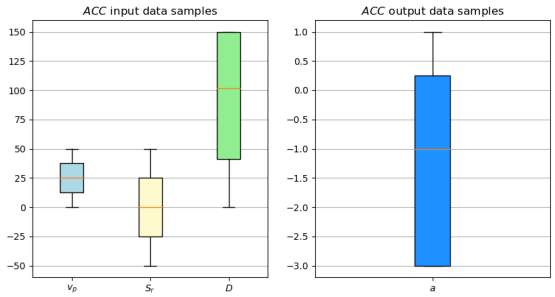
Technically, an adaptive cruise control (ACC) is an autonomous driving function of level one[1], which controls the acceleration of the *ego car* — the car whereon the ACC is installed — along the longitudinal axis. An ACC has two competing objectives: keeping the ego car at the speed set by the user (*speed following mode*) and keeping a safe distance from the *exo car* in front (*car following mode*). The ACC that we consider has one output, i.e., the acceleration a suggested to the ego car in $m \cdot s^{-2}$, and five inputs, two of which are fixed:

- $v_p[m \cdot s^{-1}]$: the speed of the ego car.
- $v_r[m \cdot s^{-1}]$: the speed of the exo car relative to the ego car; when there is no exo car, this input has the value 0.
- $D[m]$: the actual distance between the ego car and the exo car; when there is no exo car or when the exo car is farther than $150m$ this input has the default value of $150m$.
- $TH[s]$: Minimum headway time; this is the minimum time gap between the exo car and the ego car: $TH \cdot v_p$ corresponds to D_s, i.e.,the *minimum safety distance*.
- $D_0[m]$: A safety margin to be added to the minimum safety distance D_s.

In production vehicles the ACC function is implemented using classical control laws. We view the production function — called ACC_o in the following — as a black-box whose behavior should be learned by a neural network.

[1] "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles", SAE Standards, J3016_202104.

Figure 9.1 Box plot for a million samples of the Adaptive Cruise Control data set ($TH = 1.5$; $D_0 = 5$)



Given the goal of learning ACC_o using a NN, we should generate several instances of input-output data using, e.g., a car simulator. Since a simulator was unavailable to us at the time of this writing, we generated the dataset to learn various NNs by drawing samples from uniform distributions over the input values of ACC_o, considering the following lower and upper bounds for v_p, v_r and D:

$$\begin{aligned} 0 \leq v_p \leq 50 \\ -50 \leq v_r \leq 50 \\ 0 \leq D \leq 150 \end{aligned} \tag{9.1}$$

The values of TH and D_0 are kept fixed, and we obtain the corresponding output a by feeding ACC_o with the generated inputs. We generate 16 different data sets, each composed by a million samples, that feature 16 different combinations of TH and D_0, where $TH \in \{1, 1.5, 2, 2.5\}$, while $D_0 = \{2.5, 5, 7.5, 10\}$. Figure 9.1 shows the distributions of input and output samples using box plots in the case $TH = 1.5$ and $D_0 = 5$.

We tested three NN architectures comprised of affine and ReLU layers: we refer to them as *Net0*, *Net1* and *Net2* in the following. These NNs feature increasing complexity both in terms of the number of layers and in the amount of neurons per layer. The networks considered differ from one another only for the details of the hidden layers, which are the following:

- *Net0*: two affine layers of 20 and 10 neurons respectively, each followed by a ReLU layer;
- *Net1*: two affine layers of 50 and 40 neurons respectively, each followed by a ReLU layer;

- *Net2*: four affine layers of 20, 20, 20 and 10 neurons respectively, each followed by a ReLU layer.

The input of the network is in all the cases a three dimensional vector. All the networks present an output layers consisting of a linear layer of dimension 1 (without a following ReLU layer).

To learn the NNs we split the data sets in two parts, one for training and one for testing, with the ratio of 4:1. Our training phase lasts 100 epochs for each of the 16 data sets. We consider the *Adam* optimizer Kingma and Ba (2017) and the *ReduceLROnPlateau* scheduler. For both our loss function and our performance metric we leveraged the *Mean Squared Error (MSE) loss*. We set batch sizes to 32 for training, validation, and test sets. In our setup, we dedicated 30% of the training set to the validation process. Concerning the optional parameters, we also set the learning rate to 0.01, the weight decay to 0.0001 and the training scheduler patience to 3, i.e., the number of consecutive epochs without loss decrease that triggers training procedure abortion. All the training is perfomed inside NEVER2 which, in turn, is based on the PYTORCH library. For this reason, all the remaining parameters required by learning algorithms are set to their default PYTORCH values.

Verification setup. We consider three properties to be verified for the ACC case study, and we verify them in NEVER2 with different NNs. The first property that we define, called *OutBounds* in the following, simply checks that the output acceleration does not exceed the bounds of the ACC_o function. Stated formally, this amounts to have NEVER2 check that, given the preconditions in Eq. (9.1) the output a satisfies the postcondition

$$-3 \leq a \leq 1. \tag{9.2}$$

The second property we consider is called *Near0*, and it is aimed at making sure that the ACC system does not output positive accelerations when the vehicle ahead is too close. We frame this concept via the precondition

$$\begin{aligned} 0 \leq v_p \leq 50 \\ -50 \leq v_r \leq 50 \\ 0 \leq D \leq 150 \\ TH \cdot v_r + D_0 \geq D + \varepsilon \end{aligned} \tag{9.3}$$

where $\varepsilon \in \mathbb{R}^+$ is a positive tolerance value in the last inequality. Notice that the input bounds are the same as *Outbound*. The last inequality stems from the fact that $TH \cdot v_r$ is the safety distance required to stop the ego car in time if the exo car brakes, and D_0 is a buffer value which, like TH, is constant for each data set. The corresponding output postcondition for *Near0* is

$$-3 \leq a \leq 0. \tag{9.4}$$

Intuitively, we do not want the network to output positive accelerations in this case.

Finally, the last property we consider is *Far0*, which is symmetrical with respect to *Near0*. The precondition is

$$\begin{aligned} 0 \leq v_p \leq 50 \\ -50 \leq v_r \leq 50 \\ 0 \leq D \leq 150 \\ TH \cdot v_r + D_0 \leq D - \varepsilon \end{aligned} \tag{9.5}$$

where $\varepsilon \in \mathbb{R}^+$ is still a tolerance value and the input bounds coincide with *OutBounds* and *Near0* properties. In this case, we want to verify that when the ego car is too far from the exo car (or there is no vehicle ahead at all), the NN does not suggests negative accelerations. The output postcondition is

$$0 \leq a \leq 1. \tag{9.6}$$

In our experiments, we consider two different sub-settings for the mixed algorithm, called *mixed* and *mixed2* which differ in the number of neurons to refine, either 1 or 2, respectively.

9.1.2 RL-based drone hovering

Here we consider another benchmark which is based on a reinforcement learning environment: autonomous drone control. In particular, we consider the problem of making a drone take off and hover at a chosen altitude. Our motivation for dealing with a robotics framework is twofold: first, the control problems arising in robotics are more and more relevant in the real world scenario; drones and unmanned agents in general are being employed in several tasks that require a high confidence in the agent. Second, using the Soft Actor-Critic architecture we are able to employ reasonable-sized network architectures for the agent that we are able to verify, and we delegate the more complex tasks to the critic which is not required to be certified.

Figure 9.2 The Bitcraze Crazyflie 2.1 drone considered in our setup

We focused on building a modular setup for the generation of benchmarks using well-maintained and stable resources to be able to easily extend it to new case studies and network architectures. In particular, we leveraged:

- GYM[2]: an open source Python library providing a standard API for communication between reinforcement learning algorithms and environments.
- STABLE BASELINE3[3]: an open source training framework providing scripts for training and evaluating RL agents using standard state-of-the-art algorithms.
- PYBULLET[4]: an open source physics simulator for robotics and reinforcement learning.
- GYM-PYBULLET-DRONES[5]: an open source GYM-stile environment supporting the definition of various learning tasks on the control of one or more quadcopters.

Using these open source resources we greatly simplified the complexity of our setup and we were able to directly train the network of interest in the environment corresponding to our case study with the chosen state-of-the-art RL algorithm. In Figure 9.2 we show the quadcopter model of choice, which was the default one proposed in GYM-PYBULLET-DRONES (Bitcraze's Crazyflie 2.x). To evaluate our algorithms we built the experimental setup detailed in the following.

[2]https://github.com/openai/gym
[3]https://github.com/DLR-RM/stable-baselines3
[4]https://pybullet.org/
[5]https://github.com/utiasDSL/gym-pybullet-drones

Table 9.1 Actor network architectures used in our experimental evaluation, arranged in two (*AC1* to *AC4*) or three (*AC5* to *AC8*) hidden layers. The size of layers, i.e., the number of neurons in each layer, is detailed in column **No. of neurons**. Each hidden layer is followed by a ReLU layer.

Architecture	Network ID	No. of neurons
Two layers	*AC1*	32, 16
	AC2	64, 32
	AC3	128, 64
	AC4	256, 128
Three layers	*AC5*	32, 16, 8
	AC6	64, 32, 16
	AC7	128, 64, 32
	AC8	256, 128, 64

RL setup. The reinforcement learning setup we used to train our model was based on the one proposed by Panerati et al. (2021): in particular, we leveraged their HOVERAVIARY environment, together with the STABLE BASELINE3 implementation of the Soft Actor-Critic (SAC) algorithm, to train our neural networks of interest. We chose this algorithm because it is more stable than traditional actor-critic algorithms, but it still makes use of two different networks: the *actor* to learn a policy and the *critic* to approximate the optimal value function. Because of this, it is possible to learn a relatively small actor network, which is then subject to verification, while the critic network can be as complex as the task requires without impacting on the verification performances since only the actor network is relevant for this purpose. The hyperparameters chosen for the SAC algorithm were the default ones proposed by STABLE BASELINE3 except for the network architectures of the actor and the critic: in particular, for the critic we chose a fixed architecture with four hidden layers with $512, 256, 128, 64$ neurons, respectively, each followed by a ReLU layer, whereas for the actor we considered the eight different architectures presented in Table 9.1.

The observation type considered was the kinematic information (pose, linear and angular velocities) of the quadcopter, and the action type was the revolutions per minute (RPMs) applied to all the four rotors of the drone (similarly to what is done in the experimental evaluation of Panerati et al. (2021)). All the actor models considered in our experiment were trained for 50000 steps and, at the end of the training process, the version of the actor model presenting the best mean reward in an evaluation environment was chosen.

Verification setup. The verification task we consider in our experiments is an analysis of the local robustness of our actor model with respect to small variations of the input, which could be interpreted as small noise on the sensors providing the input signal. Formally we consider the following assumption:

$$\forall \varepsilon, x_0 : \quad |x - x_0|_\infty \leq \varepsilon \quad \rightarrow \quad \nu(x_0) - \delta \leq \nu(x) \leq \nu(x_0) + \delta \tag{9.7}$$

where x_0 is a specific input vector for the network ν, and ε and δ are scalar values representing the maximum noise on the input and the corresponding maximum output deviation, respectively. Our aim is to determine the values of δ corresponding to fixed values of ε and x_0 for all the actors presented in Table 9.1. The encoding of this verification task is straightforward in NEVER2: we only need to define as input star the convex polytope defined by the constraints corresponding to $|x - x_0|_\infty \leq \varepsilon$ and then propagate it in our abstract network, applying the abstract transformer presented in Section 7.2, to obtain the abstract output set, whose stars can be easily analyzed to compute the reachable bounds of the output and, as a direct consequence, the maximum value of δ. In the experiments we evaluated our complete, over-approximate and mixed algorithms: in particular, in our mixed algorithm we considered the case in which a single neuron is refined for each ReLU layer. We also considered two different values (0.1, 0.01) for ε.

Our experimental setup is available online in the public repository of PYNEVER[6], and the experiments herewith presented can be easily replicated. The experiments were run on a machine with 2 Intel Xeon Gold 6432 CPUs and 128 GB of DDR4 RAM.

[6] https://github.com/NeVerTools/pyNeVer/tree/main/examples/submissions/IEEEAccess2023

9.2 Experimental results

In this Section we present the evaluation of our methodology on the case studies detailed before, as well as further experiments involving the star elimination algorithm and the CEGAR algorithm presented in Sections 7.3 and 7.4.

9.2.1 ACC

We run our tests on a workstation featuring two Intel Xeon Gold 6234 CPU, three NVIDIA Quadro RTX 6000/8000 GPUs (with CUDA enabled), and 125.6 GiB of RAM running Ubuntu 20.04.03 LTS. For the sake of brevity, we are only going to report here in Table 9.2 a fraction of the experiments we ran for the data set with $TH = 1.5$ and $D_0 = 5$, considering $\varepsilon = 0$ and $\varepsilon = 20$. The results we show here are consistent with the results obtained on other data sets that we do not report. In particular, looking at Table 9.2 we can observe that:

- All the properties can be checked on all the networks in reasonable time by NEVER2: less than one minute of CPU time is required independently from the network architecture and the specific setting considered.
- The complete setting is the most expensive in computational terms; given the considerations above this should come at no surprise, but in one case, namely *Net2* on property *Near0*, the complete setting is able to prevail over the others, i.e., it certifies that the property is true; indeed mixed and over-approximated settings (shortened as over-approx in Table 9.2) take less time, but state that the property is false because they do not manage to reach enough precision to state the correct result.
- The over-approximated setting is often faster than the other ones: 6 out of 9 cases for $\varepsilon = 0$ and 7 out of 9 cases for $\varepsilon = 20$; however it must be noted that its results are definitive only when the property is true: 3 out of 9 cases for both values of ε and always for the (simplest) property *OutBounds*.
- The mixed setting is at times faster than the over-approximated one, but only in one case, namely property *Far0* on *Net0* it is able to provide a definite answer while outperforming both the complete and over-approximated settings.

Overall we can conclude that while further research is needed to improve on the capability of NEVER2 to provide definite answers with faster techniques involving over-approximation, still the tool is able to check a number of interesting properties in networks involving hundreds

Table 9.2 NEVER2 results for the ACC data set with $TH = 1.5$ and $D_0 = 5$, with $\varepsilon = 0$ (left) and $\varepsilon = 20$ (right). CPU time is in seconds rounded to the third decimal place. The best setting for each network and property is highlighted in boldface.

Network ID	**Property**	**Setting**	$\varepsilon = 0$		$\varepsilon = 20$	
			Result	Time	Result	Time
Net0	*OutBounds*	Over-approx	True	5.139	True	5.037
		Mixed	True	**5.055**	True	5.063
		Mixed2	True	5.112	True	**4.996**
		Complete	True	6.273	True	6.203
	Near0	Over-approx	False	5.666	False	5.034
		Mixed	False	5.251	False	5.101
		Mixed2	False	**5.203**	False	**4.965**
		Complete	False	6.319	False	5.345
	Far0	Over-approx	False	5.078	False	5.008
		Mixed	False	**4.986**	**True**	**5.016**
		Mixed2	False	5.139	True	5.068
		Complete	False	5.186	True	5.62
Net1	*OutBounds*	Over-approx	True	**5.931**	True	**5.948**
		Mixed	True	6.662	True	6.934
		Mixed2	True	7.309	True	7.232
		Complete	True	51.683	True	52.318
	Near0	Over-approx	False	**5.906**	False	**5.436**
		Mixed	False	6.676	False	5.797
		Mixed2	False	8.071	False	5.955
		Complete	False	50.469	False	7.667
	Far0	Over-approx	False	**5.709**	False	**5.344**
		Mixed	False	5.888	False	5.776
		Mixed2	False	6.301	False	6.226
		Complete	False	13.041	False	8.212
Net2	*OutBounds*	Over-approx	True	**9.525**	True	**9.532**
		Mixed	True	10.482	True	10.149
		Mixed2	True	12.525	True	12.065
		Complete	True	26.958	True	26.794
	Near0	Over-approx	False	**9.515**	False	9.379
		Mixed	False	10.292	False	9.872
		Mixed2	False	13.636	False	11.653
		Complete	False	24.496	**True**	**10.696**
	Far0	Over-approx	False	**9.753**	False	**9.453**
		Mixed	False	9.944	False	9.848
		Mixed2	False	12.148	False	11.558
		Complete	False	13.27	False	10.854

of neurons in a relatively small amount of CPU time. We view this as a positive result and an enabler for preliminary testing of NEVER2 at industrial settings featuring networks of comparable size to our ACC case study.

Table 9.3 NEVER2 results for the drones case study. Column **Network ID** refers to the same actor architectures as detailed in Table 9.1. Column **Return** reports the best return obtained during the testing of the Actors in the evaluation environment, while column **Epsilon** reports the ε values tested in our experiments. Columns ***Over-approx***, ***Mixed*** and ***Complete*** refer to the selected verification algorithm, with the maximum *Delta* (δ) obtained and the elapsed *Time* in seconds, respectively. The cells reporting "–" correspond to experiments in which our algorithm was not able to complete the verification successfully in less than 70 seconds.

Network ID	**Return**	ε	***Over-approx***		***Mixed***		***Complete***	
			δ	Time	δ	Time	δ	Time
AC1	-27.15	0.10	4.18	0.48	3.71	0.51	3.23	0.61
		0.01	0.34	0.47	0.33	0.49	0.33	0.47
AC2	-27.28	0.10	2.40	0.71	2.11	0.78	1.82	5.91
		0.01	0.10	0.52	0.10	0.52	0.10	0.58
AC3	-27.33	0.10	12.90	2.66	12.86	2.82	–	–
		0.01	1.51	0.79	1.51	0.76	1.51	1.08
AC4	-27.69	0.10	6.23	5.37	5.91	8.60	–	–
		0.01	0.41	1.27	0.40	1.58	0.40	11.84
AC5	-28.31	0.10	17.65	0.82	17.09	0.83	14.40	2.25
		0.01	1.20	0.73	1.20	0.74	1.20	0.76
AC6	-27.63	0.10	1.99	0.94	1.79	0.98	1.57	7.79
		0.01	0.11	0.77	0.11	0.78	0.11	0.78
AC7	-32.06	0.10	2.89	1.84	2.45	2.02	2.24	61.81
		0.01	0.23	0.95	0.23	0.99	0.23	1.18
AC8	-27.81	0.10	30.14	9.50	28.97	14.45	–	–
		0.01	0.56	1.63	0.56	1.78	0.56	4.91

9.2.2 Drones

From the results reported in Table 9.3 it would seem that, at least for the task considered, an increased size for the actor network does not necessarily correspond to an increase in performance. This would seem to support our belief that, in this kind of control task, small networks yield adequate performances, and therefore are still relevant in meaningful applications. Furthermore, from the values of δ obtained by our reachability analysis, it would seem that bigger networks do not present significantly increased robustness to local perturbation and, at least in our experiments, they often result to be less robust than smaller ones. Regarding the comparison between the different reachability algorithms we notice that, as expected, the values of δ found by the complete algorithm are stricter than — or

Table 9.4 Experimental results for the star elimination algorithm on the ACAS Xu benchmark. The number of stars layer-by-layer for each network is compared between the original algorithm and the elimination-based version. All times are expressed in seconds.

Network ID	**Layer**	**Original**		**Elimination**	
		No. of stars	Time	No. of stars	Time
1_1	1	18	10.89	18	10.9
	2	49	11.36	49	11.48
	3	230	12.43	230	12.56
	4	1466	24.42	1465	24.52
	5	3894	67.02	3864	66.19
	6	31706	305	31025	303.72
1_3	1	2	10.8	2	10.72
	2	30	11.27	30	11.04
	3	146	12.21	146	12.48
	4	287	13.75	287	13.87
	5	1700	28.52	1686	28.58
	6	4121	69.78	4066	70.36
2_3	1	11	10.75	11	11.04
	2	35	10.95	35	11.15
	3	102	11.77	102	12
	4	230	13.31	230	13.59
	5	408	17.46	408	17.26
	6	2128	30.58	2115	30.75

as strict as — the ones found by the mixed algorithm, and that the same behavior can be observed between the mixed and over-approximate algorithms. As we would expect, the computational time needed by the different algorithms increases with their complexity and the only exceptions appear to be when the coarser algorithms are already good enough to compute a very strict δ, which means that the number of unstable ReLU — causing the loss of precision and the splitting of the stars in the over-approximate and complete algorithm respectively — is extremely limited.

9.2.3 Star elimination

Here we show some preliminary results on the star elimination algorithm detailed in Section 7.3. For the comparison, we considered three networks from the ACAS Xu evaluation Katz et al. (2017) and the networks for drone control presented before. ACAS Xu is

Table 9.5 Experimental results for the star elimination algorithm on the drone case study, both with $\varepsilon = 0.01$ and $\varepsilon = 0.1$. The number of stars layer-by-layer for each network is compared between the original algorithm and the elimination-based version. All times are expressed in seconds.

Network ID	**Layer**	$\varepsilon = 0.01$				$\varepsilon = 0.1$			
		Original		Elimination		Original		Elimination	
		Stars	Time	No. of stars	Time	No. of stars	Time	No. of stars	Time
AC1	1	17	11.67	17	11.35	520	16.67	518	17.13
	2	25	11.34	25	11.33	2660	22.84	2579	25.01
AC2	1	3	11.13	1	11.23	3187	56.21	2712	72.09
	2	5	11.15	1	11.09	6397	119.24	5367	103.82
AC3	1	18	11.90	5	11.63	-	-	5008	228.73
	2	81	12.77	14	11.43	-	-	-	-
AC4	1	226	24.54	226	24.70	-	-	-	-
	2	490	39.52	487	43.48	-	-	-	-
AC5	1	3	11.03	3	11.06	1962	20.24	966	19.34
	2	5	11.08	5	11.10	4677	42.71	1993	25.29
	3	5	11.00	5	11.10	11318	49.97	4572	27.07
AC6	1	4	11.08	4	11.21	5992	105.60	3504	94.00
	2	4	11.06	4	11.15	11593	198.89	6287	128.23
	3	4	11.11	4	11.03	-	-	14298	171.42
AC7	1	16	11.32	16	11.38	-	-	-	-
	2	43	11.87	43	11.92	-	-	-	-
	3	88	12.14	87	12.03	-	-	-	-
AC8	1	12	12.30	10	12.94	-	-	-	-
	2	18	12.41	14	12.07	-	-	-	-
	3	45	11.89	30	11.62	-	-	-	-

an airborne collision avoidance system based on DNNs whose purpose is to issue advisory commands to an autonomous vehicle (ownship) about evasive maneuvers to be performed if another vehicle (intruder) comes too close. In particular, we selected Property 3 and 4 since they could be easily expressed as a single verification query in our tool. In the words of Katz et al. (2017), these safety properties "*deal with situations where the intruder is directly ahead of the ownship and state that the NN will never issue a COC (clear of conflict) advisory*". Considering the analysis in Katz et al. (2017), each property can be assessed on 42 different networks depending on the choice of two parameters, i.e., the the previous advisory value and the time to loss of vertical separation.

Table 9.4 shows the results of the elimination algorithm for three networks in the ACAS Xu set. We selected only three networks as they are representative enough for the analysis

herewith presented. Column "**Layer**" refers to the layer of the network in which we analyze the abstract propagation: each network in the ACAS Xu pool is made of six layers and in each layer the number of stars increases. Comparing the original algorithm and the optimized one, we can see that in this case the number of stars that can be deleted is almost neglectable: in network *1_1* we remove less than 700 stars in the last layer, with more than 31k stars alone. On the other hand, the comparison of CPU time is encouraging: even with the overhead introduced by Algorithm 3 we do not pay a significant toll. This allows us to determine that, at least, we can always try in principle to optimize the number of stars in the propagation.

In Table 9.5 we show a more encouraging result. We applied the same elimination algorithm to the drones case study and in this case the experiments show that star elimination can actually improve the performance of the verification algorithm. Comparing the results with $\varepsilon = 0.01$ and $\varepsilon = 0.1$ we observe that in the former case we propagate a very small number of stars: this is reasonable since a tighter input bound is more likely to be more stable. This stability makes the elimination-based version very similar to the original one with notable exceptions, e.g., in network *AC2* all the extra stars created in the original algorithm can be deleted and only one star is significative. The experiment with $\varepsilon = 0.1$ is more representative since the number of stars grows higher. Here we obtain slightly better results than ACAS Xu for networks *AC1* and *AC2*, but already in *AC3* the optimized algorithm manages to propagate the first layer within a timeout of 5 minutes where the original algorithm fails. Then, on networks *AC5* and *AC6* it manages to cut the number of stars in less than a half with a valuable speed-up in computational time, too.

Given the preliminary nature of these experiments, the discrepancy between the CPU times in Tables 9.3 and 9.5 is due to the fact that the latter experiments are run on a machine equipped with an Intel® Core™ i7-6500U dual core CPU @ 2.50GHz, featuring 8GB of RAM and running Ubuntu Linux 16.04 LTS 64 bit.

9.2.4 CEGAR

Here we provide the results of the empirical evaluation of the CEGAR method. All the experiments ran on a laptop equipped with an Intel i7-8565 CPU (8 core at 1.8GHz) and 16 GB of memory with Ubuntu 20 operating system. We test the CEGAR algorithm on the ACAS Xu benchmark: among the networks available, we selected those for which our over-approximate analysis could not find a definitive answer, ending with a total of 9 networks.

Table 9.6 Experimental results for CEGAR on a subset of ACAS Xu networks. Columns **Property** and **Network ID** report the property and the network considered, respectively. The other columns report the verification time in seconds and result (*Verified*) for **Mixed**, **CEGAR-PS** and **CEGAR-mR** analyses, respectively. Given the randomic nature of the counter-example generator, we report the average time and the number of results over 10 repetitions of the experiment.

Property	**Network ID**	**Mixed**		**CEGAR-PS**		**CEGAR-mR**	
		Time	Verified	Time	Verified	Time	Verified
# 3	*1_1*	13	True	10	3/10	9	9/10
	1_3	10	True	14	6/10	10	0/10
	2_3	7	True	10	9/10	7	6/10
	4_3	15	True	17	10/10	14	10/10
	5_1	6	True	11	10/10	9	10/10
# 4	*1_1*	11	True	10	0/10	9	0/10
	1_3	8	True	16	0/10	11	0/10
	3_2	12	True	12	10/10	12	10/10
	4_2	12	True	11	10/10	12	10/10

In Table 9.6 we show the performance of the two versions of the refinement algorithm explained in Section 7.4, and we compare them with our mixed abstraction methodology. The PS refinement (*CEGAR-PS*) selects six neurons in the whole network to refine, while the mR refinement (*CEGAR-mR*) refines one single neuron for each layer. Note that, by design, the number of neurons refined is the same for every methodology: six in the whole network. The difference between the three algorithms is which neurons are selected and how. As can be seen, the performances of the two refinement algorithms are comparable; however, they seem to be less effective than our mixed methodology and CEGAR-PS seems to be slightly more accurate than CEGAR-mR at the cost of a small increase in the time needed to solve the query. We believe that the difference in performance is mainly attributable to the fact that, while the measurements of relevance we used are valid, they do not capture how the coarseness of the abstraction changes dynamically when a particular neuron is refined. On the contrary, the mixed methodology chooses in each layer the neuron to refine based on the values of the areas of the triangles given the previous layer output. As a consequence, the choice of which neurons to refine is guided by the coarseness of the abstraction *after* the refinement is already applied in the previous layers.

Chapter 10

Conclusions and Future work

10.1 Conclusions

The work presented in this Thesis has advanced the state of the art with the following contributions:

- We experimented with declarative encodings for the automated design of a complex system, and generalized some best practices that can apply to different case studies.
- We expanded and structured the reachability analysis for DNNs employing the notion of generalized Star set.
- We developed some variations of the verification algorithms pushing towards the optimization of some existing bottlenecks in the procedures.
- We developed state of the art tools for both case studies, i.e., a web-based application for the design of elevator systems and two GUIs for the construction, conversion, learning and verification of DNNs.

In Demarchi et al. (2019) we started by encoding the elevator design problem first described in Annunziata et al. (2017) using the Satisfiability Modulo Theories paradigm, and we observed that with the employ of specific constraints it was possible to reach and also outperform the baseline algorithm. Encouraged by this result, we considered using pure Constraint Programming solvers in Demarchi et al. (2021) in order to understand the impact of the encoding choiches that we observed with SMT. While a full encoding is still best handled by an SMT solver when encountering computations involving real parameters, CP solvers provided a significant speed-up when considering integer encodings only. Finally, in Cicala et al. (2022) we resumed all our work and included another comparison using Genetic Algorithms.

We applied the work of Guidotti et al. (2020a) and Guidotti et al. (2021) to new case studies, namely Adaptive Cruise Control in Demarchi et al. (2022) and Reinforcement Learning-based drone control. Here we perfectioned our tool NEVER2 in order to build the networks, train them and define the properties to verify. The drone setup is the first step for a RL-based verification framework for robotics case studies that we are building. We also investigated an optimization for parts of our verification methodology in order to cope with the pervasive and well-known scalability issues: in Demarchi and Guidotti (2022) we investigated a counter-example guided abstract refinement which was introduced in Guidotti (2022) and we experimented with a way for deleting degenerate stars in the complete verification algorithm.

Finally, we published state of the art tools in both domains. LIFTCREATE is a research prototype available at liftcreate.ailift.it, still focused on the heuristic engine but designed in order to be able to switch between new engines on the fly. COCONET and NEVER2 are the two interfaces in the NeVerTools portfolio, which is part of a larger open-source ecosystem which is still in progress, called *NeuralVerification*[1]: here we collect all the resources about our research and development including the VNN-LIB standard Guidotti et al. (2023) that we use in the verification community.

In conclusion, the work presented in this Thesis can be evaluated on the initial research questions:

- (i) The choice of specific encodings, i.e., arithmetic theories, logic encapsulation, optimization, is a key point in the design encoding. In fact, the experimental evaluation proves how impactful they can be in order to obtain a result faster than other methodologies.

- (ii) The choice of solvers and tools is complementary to the choice of the encoding: there is no silver bullet when it comes to choose a tool for solving a design problem. Depending on the problem shape and how it is encoded, different solvers yield different results.

- (iii) The integration of new encodings in an existing and complex application was a challenging task that impacted the original architecture as well. In fact, in order to factorize common elements in the design flow and to build design processes that could be seamlessly integrated in the base application, it was necessary to rewrite or refactor several tasks that degraded the baseline performances, too.

- (iv) Abstraction-based methods for the verification of neural networks have proven very efficient in the compact representation of sets for evaluating the reachability of a network. On the other hand, exact analysis suffers from the curse of dimensionality when it is necessary to branch into several alternatives in order to explore exhaustively the solutions space. Corroborated by different case studies, this Thesis proposes an improvement of exact reachability analysis by discarding duplicate solutions during branching.

- (v) The most important contribution that is presented in this Thesis is the *NeVerTools* suite with the tools COCONET and NEVER2. In fact, NEVER2 is the only tool in the

[1] www.neuralverification.org

verification community to provide a single environment where it is possible to build, edit, learn and verify a network. Leveraging the VNN competition, the employ of CoCoNet and NeVer2 by practitioners interested in providing guarantees on their systems should benefit from contributions by the whole community.

10.2 Future work

Here we outline the research directions that could follow the work of this Thesis. The promising results obtained experimenting with LIFTCREATE and declarative encodings have proven valuable enough to consider a commercial distribution of the tool for technical designers.

Our experimental analysis allows us to also make considerations in a broader sense, reasoning about design of technical systems *in general*. Considering the current research and industry scenario, where implementation of algorithms, tools, solvers and computational capability are a widespread available resource, possible and viable extension to this research can be the study of techniques to optimize the problem definition and encoding. Professional engineers strive for a computational approach to design but, except for some very advanced solutions, tools are somehow related to technical legacies or to partial, non-integrated solution.

For what concerns the topic of neural networks verification, there are several things that could improve the functionalities and capabilities of the application. While PYNEVER is structured to support the conversion and training of sequential network architectures, the verification methodology only supports fully connected layers with ReLU and Logistic activation functions: for this reason our principal interest is to provide abstract transformers for other layers in order to be able to work with more complex architectures. To this end, we are interested in constraint propagation as a mean to provide a faster approximate algorithm. Preliminary tests using plain constraint propagation on fully connected architectures showed that it is possible to propagate bounds with any activation function in a fraction of the time we use in our abstraction, altough with a very coarse approximation. Employing further constraints, specific for each layer, could help use improve the results.

Finally, the problem of scalability remains the principal issue in every computer-intensive application and we can only try to optimize the search space or to employ dedicated LP solvers for the greater bottlenecks — which are the bounds computation in all algorithms. We are also working on improving the CEGAR algorithm for refining approximate analysis and we are deploying new RL-based case studies that could help us to benchmark our future implementations.

References

Abdi, H., Valentin, D., and Edelman, B. (1999). *Neural networks*. Number 124. Sage.

Adobe (2022). Snap.svg - the JavaScript SVG library for the modern web, http://snapsvg.io/.

Akintunde, M., Lomuscio, A., Maganti, L., and Pirovano, E. (2018). Reachability analysis for neural agent-environment systems. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, pages 184–193. AAAI Press.

Annunziata, L., Menapace, M., and Tacchella, A. (2017). Computer Intensive Vs. Heuristic Methods In Automated Design Of Elevator Systems. In *European Conference on Modelling and Simulation, ECMS 2017, Budapest, Hungary, May 23-26, 2017, Proceedings*, pages 543–549.

Ascione, F., Bianco, N., De Stasio, C., Mauro, G. M., and Vanoli, G. P. (2016). Simulation-based model predictive control by the multi-objective optimization of building energy performance and thermal comfort. *Energy and Buildings*, 111:131–144.

Bacchus, F. (2007). GAC Via Unit Propagation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP-2007)*, pages 133–147.

Bai, J., Lu, F., and Zhang, K. (2023). ONNX: Open Neural Network Exchange, https://github.com/onnx/onnx.

Bak, S. and Duggirala, P. S. (2017). Simulation-equivalent reachability of large linear systems with inputs. In *International Conference on Computer Aided Verification*, pages 401–420. Springer.

Barret, C., Fontaine, P., and Tinelli, C. (2017). The SMT-LIB standard - version 2.6, http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

Barrett, C. and Tinelli, C. (2018). Satisfiability Modulo Theories. In *Handbook of Model Checking*, pages 305–343. Springer.

Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009). Satisfiability modulo theories. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press.

Benavides, D., Trinidad, P., and Cortés, A. R. (2005). Using constraint programming to reason on feature models. In *SEKE*, volume 5, pages 677–682.

Brailsford, S. C., Potts, C. N., and Smith, B. M. (1999). Constraint satisfaction problems: Algorithms and applications. *European journal of operational research*, 119(3):557–581.

Brown, D. C. (1998). Defining Configuring. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):301–305.

Carlson, S. E. (1996). Genetic algorithm attributes for component selection. *Research in Engineering Design*, 8(1):33–51.

Chang, K.-H. (2015). Chapter 19 - multiobjective optimization and advanced topics. In Chang, K.-H., editor, *e-Design*, pages 1105 – 1173. Academic Press, Boston.

Chu, G. (2013). Improving combinatorial optimization. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

Cicala, G., Demarchi, S., Menapace, M., Annunziata, L., and Tacchella, A. (2022). A comparison of declarative ai techniques for computer automated design of elevator systems. *Intelligenza Artificiale*, 16(1):131–150.

Davis, L. (1991). *Handbook of genetic algorithms*. CUMINCAD.

de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg. Springer Berlin Heidelberg.

Demarchi, S. and Guidotti, D. (2022). Counter-example guided abstract refinement for verification of neural networks. In López, R. L., Quintín, D. M., Palumbo, F., Pilato, C., and Tacchella, A., editors, *Proceedings of the CPS Summer School PhD Workshop 2022 co-located with 4th Edition of the CPS Summer School (CPS 2022), Pula, Sardinia (Italy), September 19-23, 2022*, volume 3252 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Demarchi, S., Guidotti, D., Pitto, A., and Tacchella, A. (2022). Formal verification of neural networks: A case study about adaptive cruise control. In Hameed, I. A., Hasan, A., and Alaliyat, S. A., editors, *Proceedings of the 36th ECMS International Conference on Modelling and Simulation, ECMS 2022, Ålesund, Norway, May 30 - June 3, 2022*, pages 310–316. European Council for Modeling and Simulation.

Demarchi, S., Menapace, M., and Tacchella, A. (2019). Automating Elevator Design with Satisfiability Modulo Theories. In *IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, Oregon, November 4-6, 2019, Proceedings*.

Demarchi, S., Menapace, M., and Tacchella, A. (2021). Automated Design of Elevator Systems: Experimenting with Constraint-based approaches. In *The 20th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2021), Online Event December 1-3, 2021, Proceedings*.

Dutta, S., Chen, X., Jha, S., Sankaranarayanan, S., and Tiwari, A. (2019). Sherlock - A tool for verification of neural network feedback systems: demo abstract. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 262–263.

Falkner, A., Haselböck, A., Schenner, G., and Schreiner, H. (2011). Modeling and solving technical product configuration problems. *AI EDAM*, 25(2):115–129.

Franke, D. W. (1998). Configuration research and commercial solutions. *AI EDAM*, 12(4):295–300.

Franzle, M., Herde, C., Teige, T., Ratschan, S., and Schubert, T. (2007). Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236.

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. T. (2018). AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 3–18.

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2015). Explaining and harnessing adversarial examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Gu, Z., Rothberg, E., and Bixby, R. (2019). Gurobi optimization, http://www.gurobi.com/.

Guidotti, D. (2022). *Verification and Repair of Machine Learning Models*. PhD thesis, University of Genoa, Italy.

Guidotti, D., Demarchi, S., Tacchella, A., and Pulina, L. (2023). The Verification of Neural Networks Library (VNN-LIB), www.vnnlib.org.

Guidotti, D., Leofante, F., Pulina, L., and Tacchella, A. (2020a). Verification of neural networks: Enhancing scalability through pruning. In Giacomo, G. D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., and Lang, J., editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2505–2512. IOS Press.

Guidotti, D., Pulina, L., and Tacchella, A. (2020b). Never 2.0: Learning, verification and repair of deep neural networks. *arXiv preprint arXiv:2011.09933*.

Guidotti, D., Pulina, L., and Tacchella, A. (2021). pyNeVer: A framework for learning and verification of neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 357–363. Springer.

Guidotti, D., Tacchella, A., Pulina, L., and Demarchi, S. (2022). NeVer 2.0, https://github.com/nevertools/never2.

Güngör, B. (2022). Meta-Heuristical Constrained Optimization Based on a Mechanical Design Problem. *Recent Advances in Intelligent Manufacturing and Service Systems*, pages 89–99.

Hervieu, A., Marijan, D., Gotlieb, A., and Baudry, B. (2016). Practical minimization of pairwise-covering test configurations using constraint programming. *Information and Software Technology*, 71:129–146.

Homaifar, A., Qi, C. X., and Lai, S. H. (1994). Constrained optimization via genetic algorithms. *Simulation*, 62(4):242–253.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Huang, X., Kroening, D., Kwiatkowska, M., Ruan, W., Sun, Y., Thamo, E., Wu, M., and Yi, X. (2018). Safety and trustworthiness of deep neural networks: A survey. *arXiv preprint arXiv:1812.08342*.

IBM (2017). IBM ILOG CPLEX optimization studio (2017) CPLEX users manual, version 12.7.

Igel, C., Glasmachers, T., and Heidrich-Meisner, V. (2008). Shark. *Journal of Machine Learning Research*, 9:993–996.

Jensen, R. M. (2004). Clab: A c++ library for fast backtrack-free interactive product configuration. In *International Conference on Principles and Practice of Constraint Programming*, pages 816–816. Springer.

Katz, G., Barrett, C. W., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017). Reluplex: An efficient SMT solver for verifying deep neural networks. In Majumdar, R. and Kuncak, V., editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer.

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. W. (2019). The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 443–452.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

LeCun, Y., Bengio, Y., and Hinton, G. E. (2015). Deep learning. *Nature*, 521(7553):436–444.

Leofante, F., Narodytska, N., Pulina, L., and Tacchella, A. (2018). Automated verification of neural networks: Advances, challenges and perspectives. *CoRR*, abs/1805.09938.

Lv, M., Li, J., Du, H., Zhu, W., and Meng, J. (2017). Solar array layout optimization for stratospheric airships using numerical method. *Energy Conversion and Management*, 135:160–169.

Marcus, S., Stout, J., and McDermott, J. (1987). VT: An expert elevator designer that uses knowledge-based backtracking. *AI magazine*, 8(4):41–41.

Marriott, K., Stuckey, P. J., Koninck, L., and Samulowitz, H. (2014). A MiniZinc tutorial, http://www. minizinc. org/downloads/doc-latest.

McDermott, J. (1981). R1: The formative years. *AI magazine*, 2(2):21–21.

Mcdonald, K. and Prosser, P. (2002). A case study of constraint programming for configuration problems. *CiteSeerX*.

Mittal, S. and Falkenhainer, B. (1990). Dynamic constraint satisfaction. In *Proceedings eighth national conference on artificial intelligence*, pages 25–32.

Mittal, S. and Frayman, F. (1989). Towards a Generic Model of Configuraton Tasks. In *IJCAI*, volume 89, pages 1395–1401. Citeseer.

Montavon, G., Binder, A., Lapuschkin, S., Samek, W., and Müller, K. (2019). Layer-wise relevance propagation: An overview. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, volume 11700 of *Lecture Notes in Computer Science*, pages 193–209. Springer.

Müller, M. N., Brix, C., Bak, S., Liu, C., and Johnson, T. T. (2022). The third international verification of neural networks competition (vnn-comp 2022): Summary and results. *arXiv preprint arXiv:2212.10376*.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer.

Panerati, J., Zheng, H., Zhou, S., Xu, J., Prorok, A., and Schoellig, A. P. (2021). Learning to fly - a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - Oct. 1, 2021*, pages 7512–7519. IEEE.

Perron, L. and Furnon, V. (2020). OR-Tools, https://developers.google.com/optimization/.

Plaat, A. (2022). *Deep Reinforcement Learning*. Springer.

Pulina, L. and Tacchella, A. (2010). An abstraction-refinement approach to verification of artificial neural networks. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 243–257.

Pulina, L. and Tacchella, A. (2011). Never: a tool for artificial neural networks verification. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):403–425.

Pulina, L. and Tacchella, A. (2012). Challenging SMT solvers to verify neural networks. *AI Commun.*, 25(2):117–135.

Ramos, A. G., Silva, E., and Oliveira, J. F. (2018). A new load balance methodology for container loading problem in road transportation. *European Journal of Operational Research*, 266(3):1140–1152.

Rossi, F., Van Beek, P., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

Samek, W., Montavon, G., Binder, A., Lapuschkin, S., and Müller, K. (2016). Interpreting the predictions of complex ML models by layer-wise relevance propagation. *CoRR*, abs/1611.08191.

Schimpf, J. and Shen, K. (2012). ECLiPSe – from LP to CLP. *Theory and Practice of Logic Programming*, 12(1-2):127–156.

Sebastiani, R. and Tomasi, S. (2012). Optimization in SMT with LA(Q) cost functions. In *Automated Reasoning: 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings 6*, pages 484–498. Springer.

Sebastiani, R. and Trentin, P. (2018). OptiMathSAT: A tool for Optimization Modulo Theories. *Journal of Automated Reasoning*.

Singh, G., Gehr, T., Püschel, M., and Vechev, M. T. (2019). Boosting robustness certification of neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Slowik, A. and Kwasnicka, H. (2020). Evolutionary algorithms and their applications to engineering problems. *Neural Comput. Appl.*, 32(16):12363–12379.

Smith, R. L. (1996). The hit-and-run sampler: A globally reaching markov chain sampler for generating arbitrary multivariate distributions. In Charnes, J. M., Morrice, D. J., Brunner, D. T., and Swain, J. J., editors, *Proceedings of the 28th conference on Winter simulation, WSC 1996, Coronado, CA, USA, December 8-11, 1996*, pages 260–264. IEEE Computer Society.

Summerfield, M. (2007). *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming (paperback)*. Pearson Education.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., and Fergus, R. (2014). Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Taigman, Y., Yang, M., Ranzato, M., and Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 1701–1708.

The Vaadin Team (2019). *Book of Vaadin: Vaadin 14 Edition*. Vaadin.

Tjeng, V., Xiao, K. Y., and Tedrake, R. (2019). Evaluating robustness of neural networks with mixed integer programming. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Tran, H.-D. (2020). *Verification of Learning-enabled Cyber-Physical Systems*. PhD thesis, Vanderbilt University.

Tran, H.-D., Lopez, D. M., Musau, P., Yang, X., Nguyen, L. V., Xiang, W., and Johnson, T. T. (2019). Star-based reachability analysis of deep neural networks. In *International Symposium on Formal Methods*, pages 670–686. Springer.

Tran, H.-D., Yang, X., Lopez, D. M., Musau, P., Nguyen, L. V., Xiang, W., Bak, S., and Johnson, T. T. (2020). NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. *CoRR*, abs/2004.05519.

Wang, S., Pei, K., Whitehouse, J., Yang, J., and Jana, S. (2018). Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 6369–6379.

Wilhelmstötter, F. (2022). Jenetics - Java Genetic Algorithms Library, http://jenetics.io.

Wu, M., Wicker, M., Ruan, W., Huang, X., and Kwiatkowska, M. (2018). A game-based approximate verification of deep neural networks with provable guarantees. *CoRR*, abs/1807.03571.

Yeniay, Ö. (2005). Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and computational Applications*, 10(1):45–56.

Yu, D., Hinton, G. E., Morgan, N., Chien, J., and Sagayama, S. (2012). Introduction to the special section on deep learning for speech and language processing. *IEEE Trans. Audio, Speech & Language Processing*, 20(1):4–6.

Zhang, L. L. (2014). Product configuration: a review of the state-of-the-art and future research. *International Journal of Production Research*, 52(21):6381–6398.

Zheng, Y. (2019). Computing bounding polytopes of a compact set and related problems in n-dimensional space. *Computer-Aided Design*, 109:22–32.