AI for Computer Automated Design

Francesco Grella - Universitá degli Studi di Genova

May 2019

1 Introduction

The aim of the project 'AI for Computer Automated Design' for the Artificial Intelligence course was to integrate an SMT solver with the **LiftCreate** software in order to support the design of one particular class of elevators.
LiftCreate is a tool developed by Universitá degli Studi di Genova which provides an efficient and automatized design of mechanical plants, specifically elevators, which are physically described through an ontology which yields informations about many types of elevator types, for instance the distinction between hydraulic and roped. The design is performed by using parameters of commercial components stored in a database.

The scope of the project is restricted to the class of 'One Piston Hydraulic Elevator', the objective is to provide an SMT solving functionality to find optimal design configurations of components.
The placement of the components is handled geometrically, by considering the cartesian positioning of mainly three components: the **Car Frame** and the two components, **Car Door** and **Landing Door**, which are coordinated.
In particular, the position of a component is fully described by the coordinates of its high-left vertex inside the system of coordinates, whose origin is the high-left vertex of the shaft, the rectangular-sectioned space which will contain the elevator and allow its motion.

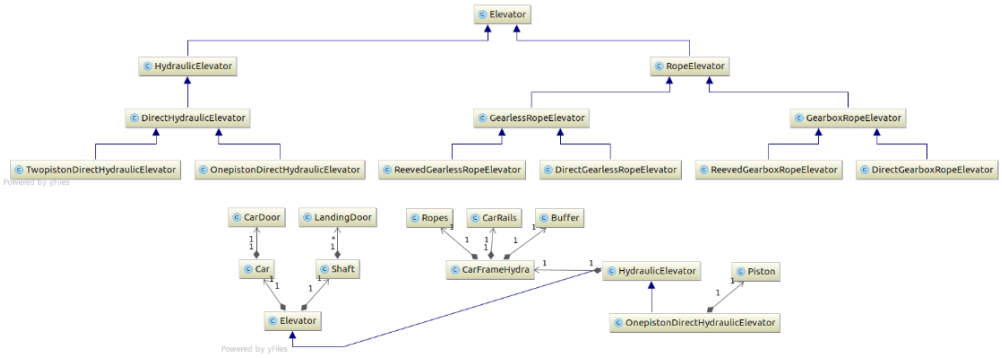


Figure 1: Class diagram of the ontology which describes the elevators in LiftCreate [1].

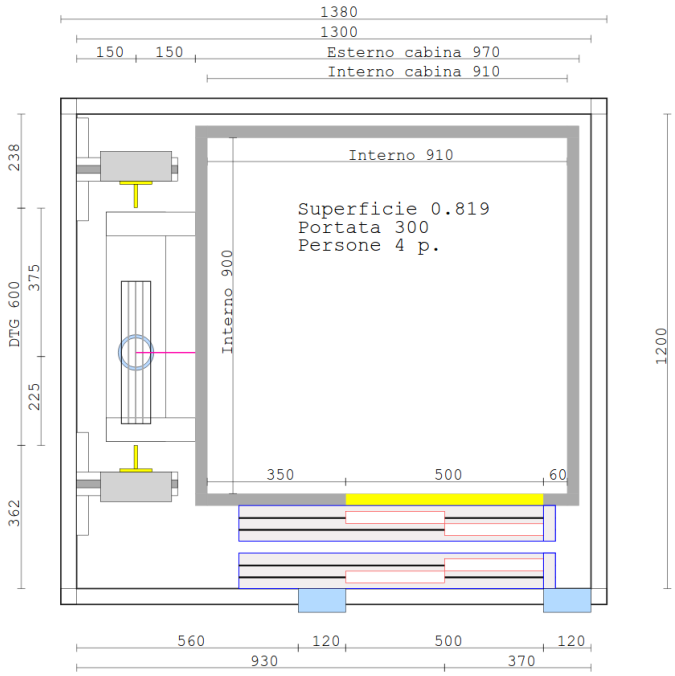


Figure 2: Sample visualization of a feasible project of Hydraulic single-roped elevator [1].

2 SMT in Artificial Intelligence

SMT is an acronym which stands for Satisfiability Modulo Theories: a SMT instance is a formula in First-Order Logic, so it provides a descriptive language which expands and generalizes the problem modeling capabilities provided by the Propositional Logic and SAT [2].
Basically the solution is found by evaluating the satisfiability of a formula in which one can put predicates, like for instance **linear inequalities**, which in the optics of this project are the preferred way of modeling the geometric constraints for the placement of the components.

Due to the versatility of the method, SMT is widely used in many Computer Science and Engineering fields, for instance in software testing, logic circuits verification or model checking and validation of hybrid systems; the functionalities of an SMT solver allow the construction of models suitable for **constraint programming**, which is the approach used in the project.
In particular the SMT solver must find feasible solutions of component placement in order to obtain a set of elevator projects starting from the descriptive parameters of single components, by solving a model of linear equality and inequality constraints based on the geometry of the problem.
Apart from the information on feasibility, the solver returns for each feasible configuration the coordinates of the characteristic vertex of the component.

SMT solvers are supported by frameworks which allow a user-friendly and high-level interaction which, like in the case of Z3, let the programmer interact with them through dedicated APIs which are specific for many languages like Python or Java.

3 Software Tools and Technologies

The developement of the project required the usage of several software technologies, both for interfacing the SMT solver with the LiftCreate software and to get objects from or publish processed data to it, even in order to obtain a visual reply of the design schemes.

- **Java** language has been used for the implementation of the module, while the environment was the professional IDE **JetBrains IntelliJ**.
- The SMT solver chosen is **Z3**, a theorem prover developed by Microsoft, which is used to check satisfiability of logical formulas. Is a low level tool which can be accessed through dedicated APIs, mostly developed for Python and Java languages. The Z3 input format extends the **SMT-LIB 2.0** standard format [3], [4].

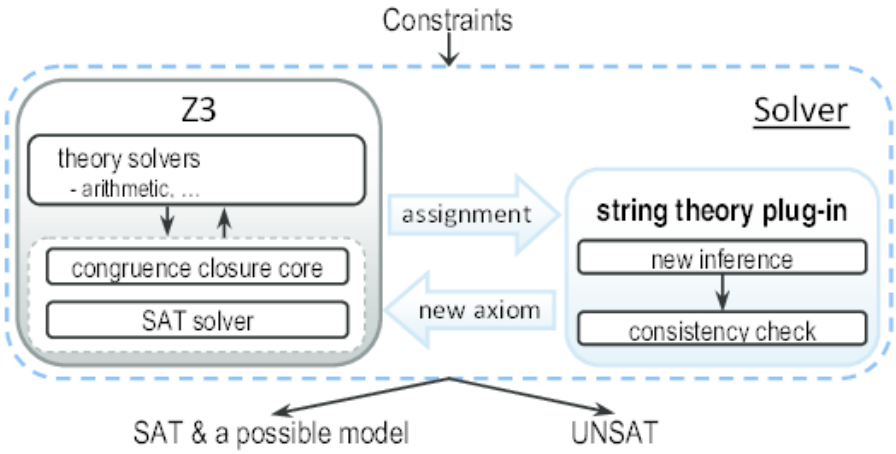


Figure 3: General architecture of the Microsoft Z3 solver: the previously built constraints pass together through the inner mechanisms of the solver and returns the result of the satisfiability check.

- The Java API for building the interface between LiftCreate and the solver is **JavaSMT**, which supports many different solvers like MathSAT, OptiMathSAT and of course Z3.
  It's developed by the Software and Computational Systems Lab (SoSy Lab) of Munchen University and published under Apache-2 Licence [5].
  It provides a framework with lots of libraries and methods to allow the user to handle, with a proper syntax, all the formulas needed for the representation of constraints in an application and simply add them to sets. Also provides intuitive methods for checking the satisfiability of the formulas previously mentioned, in this particular case after merging them all into a single formula.

- The **Spring Framework** has been used to implement the module, which utilizes the HTTP REST protocol: with this technology the module is allowed to the transmission methods GET, POST, PUT and DELETE, and so to send and receive **JSON** format messages as form of communication with the LiftCreate application [6].

- The **Vaadin** framework has been used as marginal part of the project for developing a simple web interface which allows the user to insert the desired dimensions for the shaft in which the elevator would be fitted through the spatial constraints.
  Vaadin is an open source framework written in Java and distributed under Apache 2.0 license, very useful for the software design of graphical user

interfaces: it supports **event handling programming** and **widgets**, and also the usage of graphical themes like the **"valo"** theme used in this project [7].

4 Project Developement

4.1 Early Steps

The first steps of the project have been voted to the integration and testing of the Z3 solver, its Java compliant API and the classes of LiftCreate, in particular the ones related to the design of One Piston Roped Hydraulic Elevators, which is the only type of elevator considered in the scope of the work.
The solver has been tested in solving simple formulas composed by linear equalities or inequalities, for instance the following two equations, given three variables **A**, **B**, **C**

$$A + B = C \tag{1}$$

$$A + C = 2B \tag{2}$$

can be modeled as an **OR** formula like this one:

$$(A + B = C) \vee (A + C = 2B) \tag{3}$$

After passing this trivial formula with two constraints to the solver, it will return an information on wether it's satisfiable or not; obviously in this case the formula is satisfiable and by using the **.evaluate** method on the model which contains the formulas the user can retrieve the numerical value of the first feasible assignment of the three variables.

As first interaction between the solver and the LiftCreate application, and for testing the process of getting an elevator, applying the SMT solving and resending data to obtain the modified drawing, an already existing project has been imported and some simple constraints for modifying a bit the geometry of the components were implemented, with positive results.

4.2 Constraints Implementation

The most relevant part of the project consisted in the implementation of SMT encoded constraints vote to describe the geometric problem of the component placement.
After a preliminar *domain filtering*, which given the dimensions of the shaft in terms of width and depth of its section is used for a discrimination of all those components which cannot fit the shaft, the constraint definition is addressed.
Before the in-depth explanation of the constraints it must be underlined that the *car* is built after the placement of *doors* and *frame* in a differential way.

4.3 Tables of Variables

For allowing the reader to a better comprehension of the following sections, which report several formulas of the constraints expressed in First Order Formulas and encoded to be processed by the Z3 solver, a descriptive table is provided.
The name of the variable is associated to the object which represents in the elevator design.

4.3.1 Variables for Car Frame Description

cnt	Integer Counter
CF_idx	Identifier of Car Frame
w_cf	Width of Car Frame
h_cf	Height of Car Frame
maxOh_cf	Maximum Overhang of Car Frame
piedeStaffa	Foot of the Bracket
GuideDist_cf	Distance on Y axis between the Base Point of the Car Frame and the Frame itself
clamps_w	Width which express the distance between axis and guide plus a term related to the width of the components
clamps_shift	Sums up some minor measures related to Car Frame
clamps_dist	Similar meaning of 'clamps_shift'
bracket_w1	Encodes the first width measure of the bracket
bracket_w2	Encodes the second width measure of the bracket
bracket_dist	Is a distance between Brackets

4.3.2 Variables for Door Pairs Description

cnt	Integer Counter
DP_idx	Identifier of Door Pair
opening	Door Pair Opening
dstep_cd	Door Step of the Car Door
axL_cd	Left Axis of the Car Door
axR_cd	Right Axis of the Car Door
dstep_ld	Door Step of the Landing Door
axL_ld	Left Axis of the Landing Door
axR_ld	Rigth Axis of the Landing Door
dstep_dist	Doorstep Distance
std_al	Door Alignment
frameW	Landing Door Frame Width

4.3.3 Car Frame Implication Constraints

This set of constraints is used for indexing and for allocation of the variables which will be used in the SMT encoding, which are computed from the numerical parameters get from the database of components.
The indexing of the components is done through the ***cnt*** variable.

$$CF\_idx = cnt \implies w\_cf = cf.getWidth() \tag{4}$$

$$CF\_idx = cnt \implies h\_cf = cf.getIngtrasvEff() \tag{5}$$

$$CF\_idx = cnt \implies maxOh\_cf = cf.getCarFrameHydra().getMax_sbalzo_laterale() \tag{6}$$

$$CF\_idx = cnt \implies piedeStaffa = cf.getCarFrameHydraBracket().getPiedeStaffa() \tag{7}$$

$$CF\_idx = cnt \implies guideDist\_cf = cf.getCarFrameHydraBracket().getH2() + cf.getCarRails().getRoundedH1() \tag{8}$$

$$CF\_idx = cnt \implies clamps\_w = cf.getCarFrameHydra().getDist\_asse\_guide\_filomuro() + cf.getCarFrameHydraBracket().getW4()/2 \tag{9}$$

$$CF\_idx = cnt \implies clamps\_shift = cf.getCarFrameHydraBracket().getH2() - cf.getCarFrameHydraBracket().getH4() \tag{10}$$

$$CF\_idx = cnt \implies clamps\_dist = 2 * cf.getCarFrameHydraBracket().getH4()) + (2 * cf.getCarRails().getRoundedH1()) + cf.getCarFrameHydra().getDtg() \tag{11}$$

$$CF\_idx = cnt \implies bracket\_w1 = cf.getCarFrameHydraBracket().getW1() \tag{12}$$

$$CF\_idx = cnt \implies bracket\_w2 = cf.getCarFrameHydraBracket().getW2() \tag{13}$$

$$CF\_idx = cnt \implies bracket\_dist = cf.getIngTrasvStaffe() \tag{14}$$

4.3.4 Door Pair Implication Constraints

Similarly the implicative constraints have been written for the Doors.

$$DP\_idx = cnt \implies opening = CD.getOpening() \tag{15}$$

$$DP\_idx = cnt \implies dstep\_cd = CD.getDoorstep() \tag{16}$$

$$DP\_idx = cnt \implies axL\_cd = CD.getAxisL() \tag{17}$$

$$DP\_idx = cnt \implies axR\_cd = CD.getAxisR() \tag{18}$$

$$DP\_idx = cnt \implies dstep\_ld = LD.getDoorstep() \tag{19}$$

$$DP\_idx = cnt \implies dstep\_dist = CD.getDoorstepsdistance() \tag{20}$$

$$DP\_idx = cnt \implies std\_al = LD.getAlignmentSTD() \tag{21}$$

$$DP\_idx = cnt \implies axL\_ld = LD.getAxisL() \tag{22}$$

$$DP\_idx = cnt \implies axR\_ld = LD.getAxisR() \tag{23}$$

$$DP\_idx = cnt \implies frameW = dp.getLD().getFrameWidth()$$

4.4 Boundary Constraints for Index Variables

The following constraints define boundaries for the values of the indexes associated to each Frame and each Door Pair, based on the dimensions of the data structures containing the objects.

$$CF\_idx \geq 0$$

$$DP\_idx \geq 0$$

$$CF\_idx \leq CFsize - 1$$

$$DP\_idx \leq DPsize - 1$$

4.5 Placement Constraints

Along the following sections the SMT-encoded constraints for the component placing are described, keeping on mind that all of them are within an AND formula.

4.5.1 Car Constraints

The following constraints regard the geometric limitations for the car, and take into account Reductions and Wall Thickness.

$$x\_car = w\_cf + pSets.getCWThickWEST() \tag{24}$$

$$y\_car = pSets.getRedNORTH() + pSets.getCWThickNORTH() \tag{25}$$

$$\begin{aligned} w\_car = (shaft.getWidth() - pSets.getRedEAST() - \\ pSets.getCWThickEAST() - pSets.getCWThickWEST()) - w\_cf \end{aligned} \tag{26}$$

$$\begin{aligned} h\_car = [(shaft.getDepth() - pSets.getRedNORTH() - \\ pSets.getCWThickNORTH() - pSets.getCWThickSOUTH()) - \\ (dstep\_cd + dstep\_dist) - dstep\_ld - std\_al] \end{aligned} \tag{27}$$

4.5.2 Car Frame Constraints

Here the placement constraints for the Car Frame are listed.

$$x\_cf = 0 \tag{28}$$

$$y\_cf \geq piedeStaffa \tag{29}$$

$$y\_cf + h\_cf + piedeStaffa \leq shaft.getDepth() \tag{30}$$

4.5.3 Doors Constraints

The three following constraints are related to the placement of the Door Pairs.

$$y\_cd = shaft.getDepth() - (dstep\_cd + dstep\_ld + dstep\_dist) - std\_al \tag{31}$$

$$y\_ld = y\_cd + dstep\_cd + dstep\_dist \tag{32}$$

$$x\_ld = x\_cd + (axL\_cd - axL\_ld) \tag{33}$$

4.5.4 In-Car Opening Constraints

In this section the constraints related to door opening limitations are provided.

$$x\_cd \leq x\_car + w\_car - (axL\_cd + opening/2) \tag{34}$$

$$x\_cd \geq x\_car + opening/2 - axL\_cd \tag{35}$$

4.6 Overhang Constraints

Here the constraints for handling limitations on the Overhang are reported. As for the Placement Constraints, all of them are parts of an AND formula.

$$y\_cf + (guideDist\_cf - y\_car) \geq 0 \tag{36}$$

$$y\_cf + (guideDist\_cf - y\_car) \leq maxOh\_cf \tag{37}$$

$$y\_car + h\_car + guideDist\_cf - (y\_cf + h\_cf) \geq 0 \tag{38}$$

$$y\_car + h\_car + guideDist\_cf - (y\_cf + h\_cf) \leq maxOh\_cf \tag{39}$$

4.7 Moving Components Constraints

Here the constraints regarding the reciprocal relations between moving parts, put together in an AND formula for the SMT solver, are listed.

4.7.1 Left Tolerance

$$x\_cd \geq pSets.getDoorTolerance() \tag{40}$$

$$x\_ld \geq pSets.getDoorTolerance() \tag{41}$$

4.7.2 Right Tolerance

$$x\_cd + axL\_cd + axR\_cd \leq shaft.getWidth() - pSets.getDoorTolerance() \tag{42}$$

$$x\_ld + axL\_ld + axR\_ld \leq shaft.getWidth() - pSets.getDoorTolerance() \tag{43}$$

4.7.3 Frame-in-Shaft

$$x\_ld + axL\_ld + opening/2 + frameW \leq shaft.getWidth() \tag{44}$$

$$x\_ld + axL\_ld - opening/2 + frameW \geq pSets.getDoorTolerance() \tag{45}$$

4.8 Collision Constraints

The last two constraints are related to the detection of possible collision between parts of the elevator.
They are combined in an OR formula.

$$x\_cd - pSets.getDoorTolerance() \geq bracket\_w2 \tag{46}$$

$$y\_cd - pSets.getDoorTolerance() \geq y\_cf + h\_cf \tag{47}$$

5 Testing and Results

For a first testing on the solution of elevator design problem by using SMT techniques a basic scenario has been considered to perform a timing analysis.

5.1 Cost Function Determination

The main problem at this point is the determination of a cost function for a better characterization of some placement combinations which can be better than others: under this light the first good practice to be considered is to have the car frame axis aligned with the car axis, to have the same lateral overhangs. The cost in this case has been identified as the norm of the difference between aforementioned axes, where CF stands for Car Frame and CAR stands for the effective car:

$$|axisY_{CF} - axisY_{CAR}| \tag{48}$$

The symmetry criteria should be used even in the case of symmetric doors with both sliding and folding panels, while in the case of asymmetric sliding doors, a good design must require that the opening should be close to the extreme point of the car.

A binary variable is introduced named as δ_{tel}, which is assigned to 1 if the current door is an asymmetric door and 0 otherwise.
Here a small pseudocode section describes what told before:

if Door[i].type == "TELESCOPIC" **then**
$Door[i] \implies \delta_{tel} = 1$
else
$Door[i] \implies \delta_{tel} = 0$
end if

So the cost function, by considering the new term, becomes the following:

$$(1 - \delta_{tel})|axisX_{CAR} - axisX_{DOOR}| + \delta_{tel}(extremeX_{CAR} - extremeX_{DOOR}) \tag{49}$$

Under this circumstance a smoothing weight to make the two different terms comparable will be required, since the equation could lead to a discontinuous value of the cost.
It must be underlined that the cost function built is not general, and it should be 'customized' for taking into account different particular components and situations, for instance maximizing the door opening for allowing disabled people to better access the elevator even on a wheelchair, or maybe trying to minimize the car frame dimension in order to reduce costs.
So there is a virtually huge amount of different contributions that could be use to 'compose' the cost function, based on the particular characteristics requested by the customer or maybe from safety standards, like a comparison between the maximum bearable car charge and the effective car frame charge (e.g. a car frame able to sustain the triple of the required weight is unnecessary).

For the following test only the opening maximization and car frame dimension minimization will be considered for the cost function, by adding the following contribution:

$$CF_{DTG} - DOOR_{OPENING} \tag{50}$$

It mus be said that the door opening is negated, as the other contributions are costs to be minimized.
Like introduced before, the presence of two absolute values in the cost function implies a non-linearity, which has to be handled properly by adding two new variables to the SMT problem in order to track the car frame cost and the symmetric doors cost, while in the asymmetric case the absolute value is not present due to the constraints on the door opening, which must remain in the car space.

$$\text{var } CF_{cost}$$
$$\text{var } DOOR_{costSym}$$

These variables are then assigned with the help of some constraints as implications:

$$axisY_{CF} - axisY_{CAR} >= 0 \implies cf_{cost} = axisY_{CF} - axisY_{CAR} \tag{51}$$

$$axisY_{CF} - axisY_{CAR} < 0 \implies cf_cost = axisY_{CAR} - axisY_{CF} \tag{52}$$

$$axisX_{DOOR} - axisX_{CAR} >= 0 \implies door_{costSym} = axisX_{DOOR} - axisX_{CAR} \tag{53}$$

$$axisX_{CAR} - axisX_{DOOR} < 0 \implies door_{costSym} = axisX_{CAR} - axisX_{DOOR} \tag{54}$$

so finally the objective becomes the following linear function:

$$mincf_{cost} + (1 - \delta_{tel})door_{costSym} + \delta_{tel}(extremeX_{CAR} - extremeX_{DOOR}) \tag{55}$$

that can be extended with an arbitrary combination of parameters in the case of a global optimization like explained before, for instance the following one:

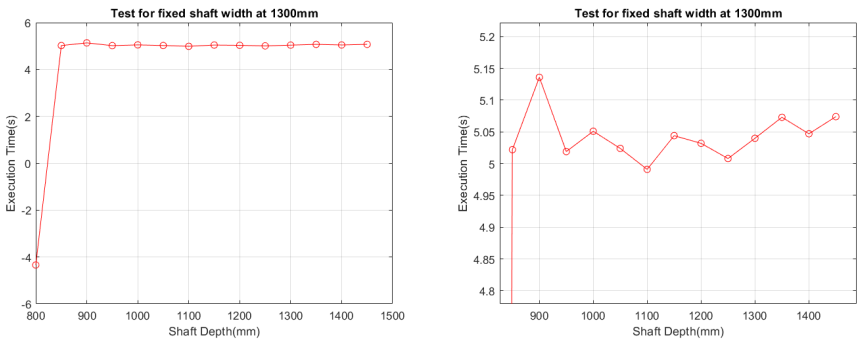
$$\begin{aligned} mincf_{cost} + (1 - \delta_{tel})door_{costSym} + \\ \delta_{tel}door_{costAsym} + CF_{DTG} - DOOR_{OPENING} \end{aligned} \tag{56}$$

it can be noticed that global contributions are numerically dominant with respect to the local ones, so normalization weights are required to flatten the difference in magnitude.
These weights are intended to be modified by the user in order to better fit the required design, by giving more importance so some features instead of others which are not of main interest for the application.
If, for example, a specific elevator is to be built in a context where it is required to grant access to disabled people, the contribution on the door opening or the car surface should be preferred to the car frame dimension one.

5.2 Timing Analysis

The simple testing scenario used for the assessment of the time required to find a solution, or notice that no solution exists, for given shaft dimensions consists of an iterative scheme with fixed width and variable depth.
In particular for 15 iterations the width has been kept fixed to *1300 mm* and for other 15 fixed to *1500 mm*, while for each of the two widths the depth is varied between *800 mm* and *1500 mm* with a step of *50 mm*.

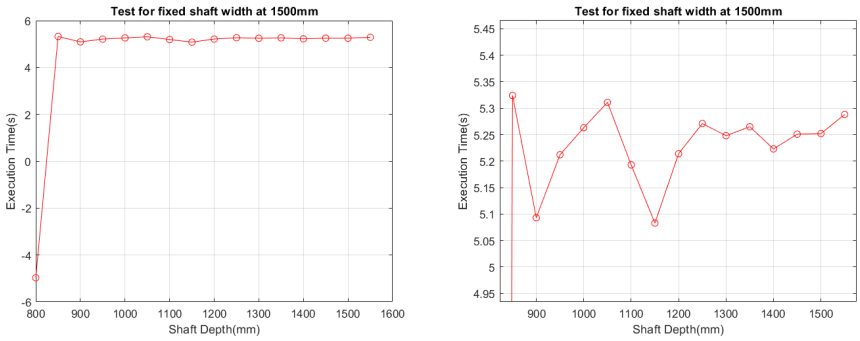
The REST call used for the tests returns the execution time in milliseconds, with a negative sign to signal when no feasible solution is found, always signaling the time needed to signal that in fact no solution is possible for the given dimensions.
The following plots shows the results of the tests, in which the cases with unfeasible solution are signed as negative time, which is not consistent, anyway more details will be provided next in this section.



(a) (b)

Figure 4: Overall and detailed plots for 1300 mm width

As shown in the first case, the overall execution time is approximately around 5 seconds, and in more in detail it never goes over 5.15 seconds.
The relevant fact that has to be signaled is that when no feasible solution is found the solver only needs 4.2 seconds to notify it.



(a) (b)

Figure 5: Overall and detailed plots for 1500 mm width

The results for the width fixed at 1500mm are similar to previous case, with an overall average time a bit increased due to the fact that greater dimensions imply more fitting components and so a broader domain of search.
In particular the bias is about 300 ms with respect to the previous considered case, and affects even the case of no feasible solution.
The last important consideration that can be done is that with the two width values provided the only case in which no solution is found is for a depth of 800 mm.

References

[1] Armando Tacchella Leopoldo Annunziata Marco Menapace. "Computer Intensive Vs. Heuristic Methods In Automated Design Of Elevator Systems". In: *31st Conference on Modelling and Simulation* (2017). DOI: https://doi.org/10.7148/2017-0543.

[2] A. Biere. *Handbook of Satisfiability*. IOS Press, 2009.

[3] Microsoft. *Z3 Prover*. URL: https://github.com/Z3Prover/z3.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[5] University of Munchen. *Software and Computational Systems Lab*. URL: https://www.sosy-lab.org/.

[6] Spring by Pivotal. *Spring Framework*. URL: https://spring.io/projects/spring-framework.

[7] Vaadin Ltd. *Vaadin*. URL: https://vaadin.com/.