Master Thesis

Automated Optimization in Production Planning

September 25, 2018

Henri Lotze

First examiner:	Prof. Dr. Erika Ábrahám
Second examiner:	Prof. Dr. Marco Lübbecke
Examiner @ Bosch:	Oliver Inkmann

Declaration of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work and has been done without inadmissable help of others. I have not used any other sources or aids other than those mentioned in this work. This thesis has not been submitted for any degree or other purposes in this or similar form. I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Selbstständigkeitserklärung

Ich versichere hiermit an Eides Statt, dass ich nach bestem Wissen und Gewissen die vorliegende Arbeit selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich versichere, dass der geistige Inhalt dieses Werkes Produkt meiner eigenen Arbeit ist und dass jegliche Quellen, welche die Erarbeitung dieser Ergebnisse unterstützt haben, vollständig genannt wurden.

Ort, Datum	Unterschrift

Abstract

In this work, we solve a planning problem for the Robert Bosch GmbH. We are given a set of orders, each representing a number of parts of a certain individual configuration that are to be produced in varying numbers over several time periods. Our task is to automatically find an assignment of these orders to a number of production lines over a horizon of time periods such that all parts of all orders can be produced. In order to be able to produce a part, a line has to have a certain configuration, which means that individual production lines have to be upgraded during production.

We are to capture the given planning problem in a mathematical form in order to produce a valid plan that is optimal or optimal to a certain degree. The degree of optimality is measured with respect to an objective function, which represents some cost of a valid plan. The directives under which our plan is evaluated and are thus part of our objective function are the utilization amount of each production line, the costs of buying so-called customer releases without an order may not be assigned to a line, the costs of upgrading and constructing production lines as well as trying to ensure that an order is produced near its delivery location and not switched around several lines over our planning horizon.

For capturing and solving the given problem, we make use of solvers for satisfiability modulo theory (SMT) as well as mixed integer programming (MIP). We detail the formulation of our objectives and our constraints in formulations for both kinds of solvers.

In order to solve this problem, we developed a software tool that is able to parse real data on the one hand as well as generate artifical data on the other hand. This data is then processed by so-called problem handlers which implement our mathematical formulations in order to find and output a valid and optimal or near-optimal plan.

Additionally, we evaluate the performance of three SMT and four MIP solvers on our given problem.

Contents

Chapter 1

Introduction

Production planning is a common problem in the industry. In its most basic form, a planner is given a set of jobs, orders or workers that are to be assigned to a set of production machines or production sites. Usually, not just any valid assignment of jobs to machines is desired, but an optimal one regarding some costs, e.g. if the machines are stationed at different sites and one wants to minimize the distance a worker has to travel to his or her workplace.
Real-world problems are usually more complex than this simple description. Machines may have limited capacities of jobs or pieces they can process per time, order sizes may change over time and additional machines may have to be built if the given capacities are not sufficient. These restrictions, combined with the desire to be able to introduce or remove constraints to the system over time make it hard and often unsustainable to write efficient special-purpose algorithms to tackle these problems.
As companies are still interested in automating the process of finding a solution to these problems, general mathematical solving tools are often employed. On a very abstract level, given a compatible mathematical formulation of the problem, these tools are able to find solutions in a rather efficient way in practice. Given a function that models some inherent or artificial costs in the system as an objective, it is even possible to find a solution that is optimal with regard to this function or provably close to the optimal solution. The two most used techniques in this field are those of (linear) mixed integer programming (MIP) as well as satisfiability modulo theory (SMT).
In this work, a planning problem for the Robert Bosch GmbH was formulated and solved. The problem was that of finding a valid assignment of a set of orders to a set of production lines over several time periods, such that each order is completely produced. The result should give the planner information about how many pieces of what order are supposed to be produced on which line, which lines may have to be upgraded to which configurations and which lines may be deconstructed due to inactivity. Additionally, the planner wants to know for which orders she or he has to buy so-called “customer releases”, without which no assignment of an order to a line may be done. Under all valid assignments, one is to be found that is optimal regarding several objectives, such as minimizing the number of needed line upgrades, utilizing lines to a certain percentage of their possible capacities and ensuring that orders are not switched around production lines if not needed. We discuss these objectives in more detail in chapter 3.
The underlying task for this work is three-fold: We are to find mathematical formulations that model our problem as close as possible. Then, we are to implement a software tool that is able to extract all data necessary for solving the model and which is able to generate problem files for both SMT and MIP solvers - and solve them - and to finally investigate the performance of

our concrete mathematical formulations using different solvers.
The rest of this paper is structured as follows. First, we properly introduce and define the terms of *satisfiability modulo theory*- and *mixed integer programming*-solving and give a short overview over how they work. Then, we introduce our given planning problem in detail and construct mathematical formulations that are compatible with both types of solving. We then shortly discuss some aspects of our implementation and conclude by taking a closer look at the performance of our formulation on real data as well as on artificially generated data.

Chapter 2

Preliminaries

Before we start our modelling process, we need to discuss what we are modelling and how the underlying solution mechanisms work. We want to give the reader a short introduction to MIP- and SMT-solving and explain in which way we are to construct our constraints and objectives. We start by giving a short overview over the notation used throughout this work.

2.1 Notation

In the following chapters, we will use the following notation for variables and constants: Constants will be completely capitalized while variables will always be written in lowercase notation, i.e.$variable_o$ and $CONSTANT_{p,t}$. We may sometimes omit the concrete sets that they range over if the index makes it clear to which set we are referring to, e.g. the index of $variable_o$ ranges over the set of all our orders, O, implicitly.
In the context of MIP-solving, the values of booleans are internally represented as the integers 0 or 1, which stand for the valuations $False$ and $True$ respectively. Whenever a boolean value is multiplied with some arbitrary value v, we thus multiply v with either 0 or 1.
If not specified otherwise, we assume the values of all single-letter symbols such as n and m to be elements of $\mathbb{N}_0$.

2.2 Satisfiability Checking

The basic satisfiability problem is for a given formula φ in propositional logic to decide whether there exists an assignment $\alpha : \{p_1, p_2, \ldots, p_n\} \rightarrow \mathbb{B}$ to all of the variables p_i of φ such that φ evaluates to true under this assignment. If such an assignment exists, we call φ *satisfiable*, else *unsatisfiable*. We call a variable and its negation *literals*.

Definition 2.2.1 (Conjunctive Normal Form). A propositional formula φ of the form

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} \varphi_{i,j}$$

with literals $\varphi_{i,j}$ is in conjunctive normal form. We call each disjunction of literals in this form a clause.

For every propositional formula, there exists an equisatisfiable formula ψ in *conjunctive normal form (CNF)*, introduced in Definition 2.2.1. This transformed formula can be generated in polynomial time and linear in size of the original formula φ by using Tseitin's transformation[1]. Furthermore, we assume the reader to be familiar with the fact that *SAT* is one of the most famous NP-complete problems, i.e. we assume that no polynomial time algorithm for solving general satisfiability problems exists. While *SAT* is considered a hard problem in theory, algorithms have been developed to solve a vast range of instances efficiently in practice. One of the most famous of these heuristic algorithms and the de-facto standard in implementations is the *Davis-Putnam-Logemann-Loveland (DPLL)* algorithm[2][3] combined with *conflict-driven clause learning (CDCL)*[4][5].
The basic idea behind the algorithm is a rather efficient exploration of the space of all possible assignments. It makes use of three sub-procedures in order to solve the satisfiability problem.
The first sub-procedure is *boolean constraint propagation (BCP)*, which checks if the presented formula currently contains unsatisfied clauses with a single undecided literal, so-called *unit clauses*. If this is the case, this single literal has to evaluate to *True*, for otherwise, we would know the whole formula to be unsatisfiable. This assignment is then applied to all occurrences of the variable contained in the literal. Afterwards, the check for unit clauses is done again on the formula, until no further unit clauses are found or a contradiction occurrs.
The second procedure is that of *deciding* the values of undecided literals. How this is done in detail depends on the implementation, as one has to choose an order of the variables that are to be decided and the value that variables are assigned to.
The third and most involved of the sub-procedures of the DPLL algorithm is *backtracking* or *conflict resolution* when using CDCL. If the algorithm creates an assignment that is not satisfying through both propagating and deciding variable values, a pruning of the space of possible assignments based on the current contradiction is desirable. Without going into detail, this is achieved through the - often implicitly used - data structure of an implication graph. This graph allows the algorithm to reason about the contradiction that occurred and to learn and add new clauses to the formula. This speeds up the search process by a big margin, making the DPLL algorithm with CDCL the core of essentially all modern SAT-solvers.

2.3 Satisfiability Modulo Theory (SMT)

SAT-solving algorithms can only be used for formulae of propositional logic, which is not powerful enough to (efficiently) encode all actual computational problems that one comes across. For this reason, solvers were developed that try to reduce the computation overhead from more rich and complex logics back to propositional logic encodings in order to benefit from the good performance and completeness of SAT-solvers. Usually, a SAT-solver is used in order to decide which subsets of higher logic constraints have to be satisfied simultaneously and this set is then solved by a complete solution mechanism for this higher logic. The actual process of applying this reduction can be further divided into two strategies, *eager* and *lazy* SMT-solving, with the latter again being subdivided into *less lazy* and *full lazy*. For the purpose of this chapter, we will only look at the most commonly used strategy, namely that of *less lazy* SMT-solving.

2.3.1 Less Lazy SMT-Solving

One of the most commonly used forms of SMT-solving is the so-called *less lazy* approach, depicted in Figure 2.1. We are given some formula φ over some logic. We then use Boolean abstraction to obtain a formula that can be processed by our SAT-solver. Most primitively, this abstraction introduces a Boolean variable for each theory constraint, encoding whether it is to be satisfied

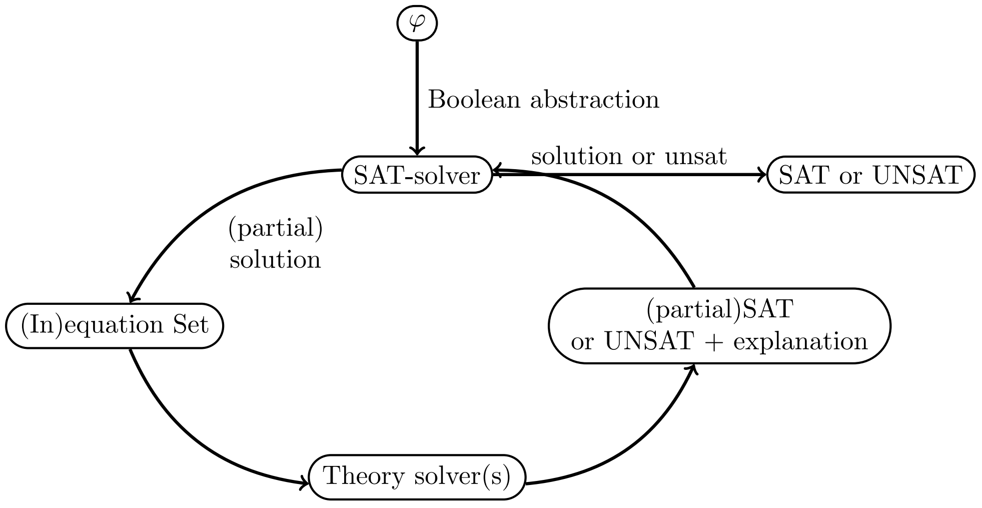


Figure 2.1: Less-Lazy SMT-Solving Process

or not. This Boolean formula is then solved, e.g. by using the DPLL algorithm. The solver does initially not completely solve the given formula, it rather applies BCP to the formula with its current (initially empty) assignment and decides on a variable if BCP did not find any new assignments. If a decision was made, BCP is applied again. Only then the current partial solution is passed on. Notice that by this procedure we only argued which constraints have to be satisfied, without checking whether the selected subset is free of contradictions regarding its theory. Thus we pass the subset of constraints to the *theory solver*, which then checks whether a satisfying assignment for the theory constraints exists. If the theory solver finds the subformula to be unsatisfiable, backtracking with conflict resolution is applied if possible. If this is not possible, we know of the formula to be unsatisfiable. If, however, the theory solver finds the subformula to be satisfiable, the SMT-solver returns SAT (if a complete assignment was made by the SAT-solver before) or the current subformula is passed back to the SAT-solver which then decides on the next unassigned variable.
Since SMT solvers are very modular in their design, they are in principle able to decide every logic for which there exists a solution mechanism. We will however see that for the problem that we are given in this work, decision procedures for linear integer arithmetic (LIA) are sufficient.

2.4 Mixed Integer Linear Programming (MIP)

The theory behind SMT-Solving has a heavy background in mathematical logic. Contrary to this, the solving of linear programs stems from a background of linear algebra. We will thus now introduce a way to model a constrained optimization problem, i.e. especially a constrained decision problem, from an algebraic view. We base the following section on the work of Schrijver[6].
From an algebraic perspective, an inequality constraint of the form $\Sigma_{i=1}^{n} a_{i,j} \cdot x_j \leq b_j$, where $a_{i,j}$ and b_j are constants and x_j is a variable, is a half-space. We can define a polytope $P \subseteq \mathbb{R}^n$ as a mathematical space induced by the cut of finitely many half-spaces, i.e. $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ with $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. If we construct a polytope this way, it is convex.
Linear optimization problems are problems that maximize or minimize the value of an *objective function* $c^T x$ with $c \in \mathbb{R}^n$ inside a polytope. We define this kind of optimization problem in

definition 2.4.1.

Definition 2.4.1 (Linear Optimization Problem)**.** Given $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m, c \in \mathbb{R}^n$. A linear optimization problem is defined as

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R} \end{aligned}$$

The area of solving problems from linear real arithmetic is well-studied and efficient general solving procedures have been proposed over the decades, with the most popularly used procedure being *Simplex*[7]. However, a majority of problems cannot be modelled using linear real arithmetic since often discrete values are wanted. An intuitive example would be that of a decision: Should an action be taken or not? For such a decision, only a valuation of *True* or *False* makes sense. Another example would be that of assigning a number of workers to a job, for only the assignment of a discrete number of workers results in a valid plan. If we want to model over a discrete space using linear integer arithmetic, we call the resulting model an *integer linear program (ILP)*. Generally, for modelling linear optimization problems over a space that is partly discrete and continuous, we use *mixed-integer programs (MIP)*, introduced in definition 2.4.2. We will see that the planning problem of this work is indeed a MIP, as it contains only continuous variables and Boolean Variables. Note that in the context of LP-solving, $\mathbb{B}$ is defined as the set $\{0, 1\}$, i.e. especially, $\mathbb{B} \subset \mathbb{Z}$.

Definition 2.4.2 (Mixed Integer Optimization Problem)**.** Given $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m, c \in \mathbb{R}^n$. A mixed integer optimization problem is defined as

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{R}, i \in \{1, \ldots, k\} \\ & x_j \in \mathbb{Z}, i \in \{k+1, \ldots, n\} \end{aligned}$$

In order for a solver to be able to process our objective and set of constraints, we will need to make sure that all our terms are linear. In particular, this excludes the multiplication of variables and the use of abstract values.

Chapter 3

Modelling

Now that we have established the general solving procedures, we turn to creating mathematical formulations out of the constraints that our problem consists of.
We are given a set of orders O which consist of a number of parts that are to be produced. We are supposed to assign these orders to a set of production lines L over some discrete time periods T. In this chapter, we sometimes abbreviate the term *production lines* by *lines*. Every production line is able to produce a certain subset of all possible products, depending on its current configuration. We denote the current configuration of a line by the term "line type" and the set of all line types by P. Lastly, production lines are located at production sites, of which the set we will call S. On an abstract level, our goal is to find a valid assignment of all orders to a set of lines over all time periods such that each order can be completely produced, while possibly upgrading or constructing new lines in the process.

In the following, we will explain the modelling of and the motivation behind each constraint in more detail. The modelling process is split into two parts: Constructing a basic, reduced model that does not capture concepts such as upgrading a line or continuity of assignments to a line and constructing a complete model. We have chosen this approach to first ensure that the given data is consistent on a smaller model and to make the process of debugging our model easier by iteratively adding the constraints of the complete formulation.
We will first mention the formulation of the reduced model and afterwards add additional information encoded by the complete model. We will mention the constraints for both the SMT and MIP formulations together or shortly one after another instead of seperating them into different sections. A reference for all variable and constant names along with descriptions can be found at the end of this chapter.

3.1 The Objective Function

Our objective consists of five optimization directions that compete with one another.
Firstly, it is given that we want production lines to be utilized around a certain threshold of a fixed percentage, which is usually around 85%. This is so that - at the end of the planning phases or during one of the discrete time periods - more capacities can be scheduled to existing production lines. To calculate the concrete percentage for each line, we are given concrete target production values in a number of pieces, as well as the maximum number of parts a production line is able to produce. We thus obtain our target percentage by simply dividing the target capacity by the maximum capacity for each line.
Secondly, we may have to obtain customer releases in order to assign orders to lines, since

customers want to retain a certain degree of control over where the parts that they have ordered are produced. We assume that the cost of obtaining a release is only dependent on the order - i.e. the customer behind it - and not the concrete line for which a release has to be bought. Obtaining a release is a matter of both money and time, and this part of the objective function is supposed to encode the actual monetary costs as well as the overhead of the process of obtaining a release. As the plan that we are trying to obtain is only realized starting a year after being made, we assume all customer releases to having been obtained by the start of production, thus not requiring additional constraints which may restrict production in the first time periods.
As a third part of the objective, we are to respect the concept of "local for local", i.e. matching the location of the production site with the location of delivery. This is rather intuitive, as shipping produced parts around the world is costly and causes a delayed delivery.
The fourth part of the objective is encoding the cost of incontinuity, i.e. moving orders from line to line over discrete time periods. This produces a multitude of costs, as not only is the customer unable to deduce why the order is moved around without reasoning, it also means managing the actual logistics and planning of these changes.
Lastly, if we want to be able to upgrade or construct production lines, we have to encode the costs of each of these processes. As we will see in this chapter, upgrading and building are similar enough for the model to be joined together in the abstract process of *upgrading.* Note that for simplification, we assume the process of upgrading itself to be free of costs, while of course the applied upgrades themselves may still cost money. This means that our objective function assumes the costs of adding two features to a line at once to be the same as adding the features in two different upgrade steps. This is further discussed in section 3.3.3. While the process of upgrading existing lines itself is assumed to be free of costs, newly constructing a line does have an inherent cost additional to the costs of newly built features.
As not all terms of the objective function can be directly measured by a monetary cost, such as e.g. splitting an order between multiple lines, all cost coefficients contain artifical cost values of similar dimensions. As priorities among the different terms of the objective function may change over time and since changing each single cost coefficient by hand is not feasible regarding the effort, we additionally multiply each cost coefficient term by a factor $\omega_1 \dots \omega_5$ to allow for an easy weighing of the objective terms against one another by the user.
For readability, we introduce the constant $TUTIL_{l,t}$ to hold the target utilization that is given by the capacity constants. We let ε be a value greater than 0:

$$\frac{CAPS_{l,t} + CAPNS_{l,t}}{CAPM_{l,t} + CAPNM_{l,t} + \varepsilon} = TUTIL_{l,t} \quad \forall l \in L, t \in T$$

If we combine these verbal descriptions, the following is a first proposition for modelling the objective function for a MIP:

$$\min \quad \sum_{t\in T}\sum_{l\in L} |TUTIL_{l,t} \cdot actv_{l,t} - \frac{\Sigma_{o\in O} prod_{o,l,t}}{CAPM_{l,t} + CAPNM_{l,t} + \varepsilon}| \cdot UTIL^{\$}_{l} \cdot \omega_1 \tag{3.10}$$

$$+ \quad \sum_{o\in O}\sum_{l\in L} buyrel_{o,l} \cdot RELEASE^{\$}_{o} \cdot \omega_2 \tag{3.11}$$

$$+ \quad \sum_{t\in T}\sum_{o\in O}\sum_{l\in L} [asgn_{o,l,t} \cdot (1 - LOCAL_{o,l})] \cdot LFORL^{\$}_{o,l} \cdot \omega_3 \tag{3.12}$$

$$+ \quad \sum_{t\in T_{>0}}\sum_{o\in O}\sum_{l\in L} |(asgn_{o,l,t-1} - asgn_{o,l,t})| \cdot CONT^{\$} \cdot \omega_4 \tag{3.13}$$

$$+ \quad \sum_{t\in T}\sum_{p\in P}\sum_{l\in L} upgr_{l,p,t} \cdot UPG^{\$}_{l,p} \cdot \omega_5 \cdot 2^{|T|-t} + newb_{l,t} \cdot BLD^{\$} \tag{3.14}$$

- (3.10) Penalty costs for not reaching the expected percentage of utilization, if the line is used. ($\varepsilon > 0$)
- (3.11) Costs for aquiring a customer release for a line.
- (3.12) Penalty costs for not honoring "local for local" when assigning an order to a production line.
- (3.13) Penalty costs for switching the production of a task to a different line.
- (3.14) Costs for building or upgrading a line to a new type. We will discuss the additional factor $2^{|T|-t}$ in the section *Challenge: The Size of P*.

Looking at the formulation, we can see that it is not in a linear form needed for a MIP or SMT formulation that we are aiming for. The objective function still utilizes absolute values, in detail, the following two parts of the objective are non-linear:

$$\sum_{t\in T}\sum_{l\in L}|TUTIL_{l,t}\cdot actv_{l,t} - \frac{\Sigma_{o\in O}prod_{o,l,t}}{CAPM_{l,t}+CAPNM_{l,t}+\varepsilon}|\cdot UTIL_l^{\$}\cdot\omega_1 \tag{3.10}$$

$$\sum_{t\in T_{>0}}\sum_{o\in O}\sum_{l\in L}|(asgn_{o,l,t-1}-asgn_{o,l,t})|\cdot CONT^{\$}\cdot\omega_4 \tag{3.13}$$

As both terms include a simple difference, we are able to linearize them by introducing a new variable for each. We make use of the following fact:

$$|p|\le q \Leftrightarrow (p\le q)\wedge(-p\le q)\quad \forall p,q\in\mathbb{R}$$

The reformulation of (3.10) is thus:

$$\begin{aligned}
\min\quad & \sum_{t\in T}\sum_{l\in L}uaux_{l,t}\cdot UTIL_l^{\$}\cdot\omega_1 && && (3.10)\\
\text{s.t.}\quad & uaux_{l,t}\ge TUTIL_{l,t}\cdot actv_{l,t}-\frac{\Sigma_{o\in O}prod_{o,l,t}}{CAPM_{l,t}+CAPNM_{l,t}+\varepsilon} && \forall l\in L,t\in T && (3.27)\\
& uaux_{l,t}\ge \frac{\Sigma_{o\in O}prod_{o,l,t}}{CAPM_{l,t}+CAPNM_{l,t}+\varepsilon}-TUTIL_{l,t}\cdot actv_{l,t} && \forall l\in L,t\in T && (3.28)\\
& uaux_{l,t}\in\mathbb{R}_+ && \forall l\in L,t\in T &&
\end{aligned}$$

With the SMT reformulation being:

$$\begin{aligned}
\min\quad & \sum_{t\in T}\sum_{l\in L}uaux_{l,t}\cdot UTIL_l^{\$}\cdot\omega_1 && && (3.10)\\
\text{s.t.}\quad & actv_{l,t}\rightarrow(uaux_{l,t}\ge TUTIL_{l,t}-\frac{\sum\limits_{o\in O}prod_{o,l,t}}{CAPM_{l,t}+CAPNM_{l,t}+\varepsilon}) && && \\
& \wedge(uaux_{l,t}\ge\frac{\sum\limits_{o\in O}prod_{o,l,t}}{CAPM_{l,t}+CAPNM_{l,t}+\varepsilon}-TUTIL_{l,t}) && \forall l\in L,t\in T && (3.27)\\
& \neg actv_{l,t}\rightarrow(uaux_{l,t}=0) && \forall l\in L,t\in T && (3.28)\\
& uaux_{l,t}\in\mathbb{R}_+ && \forall l\in L,t\in T &&
\end{aligned}$$

And, following the same reformulation scheme, (3.13) becomes:

$$\begin{aligned}
\min \quad & \sum_{t\in T_{>0}} \sum_{o\in O} \sum_{l\in L} aaux_{o,l,t} \cdot CONT^{\$} \cdot \omega_4 && && (3.13)\\
\text{s.t.} \quad & aaux_{o,l,t} \geq asgn_{o,l,t-1} - asgn_{o,l,t} && \forall o \in O, l \in L, t \in T_{>0} && (3.29)\\
& aaux_{o,l,t} \geq asgn_{o,l,t} - asgn_{o,l,t-1} && \forall o \in O, l \in L, t \in T_{>0} && (3.30)\\
& aaux_{o,l,t} \in \mathbb{B}_+ && \forall o \in O, l \in L, t \in T_{>0} &&
\end{aligned}$$

Looking at the SMT formulation of this transformed objective, one can see that we can combine both of the MIP constraints into one:

$$\begin{aligned}
\min \quad & \sum_{t\in T_{>0}} \sum_{o\in O} \sum_{l\in L} ITE(aaux_{o,l,t}, CONT^{\$} \cdot \omega_4, 0) && && (3.13)\\
\text{s.t.} \quad & aaux_{o,l,t} \leftrightarrow (asgn_{o,l,t-1} \oplus asgn_{o,l,t}) && \forall o \in O, l \in L, t \in T_{>0} && (3.29, 3.30)\\
& aaux_{o,l,t} \in \mathbb{B}_+ && \forall o \in O, l \in L, t \in T_{>0} &&
\end{aligned}$$

The complete objective function for the MIP formulation with the initially mentioned weighing factors thus becomes:

$$\begin{aligned}
\min & \sum_{t\in T} \sum_{l\in L} uaux_{l,t} \cdot UTIL_l^{\$} \cdot \omega_1 && (3.10)\\
+ & \sum_{o\in O} \sum_{l\in L} buyrel_{o,l} \cdot RELEASE_o^{\$} \cdot \omega_2 && (3.10)\\
+ & \sum_{t\in T} \sum_{o\in O} \sum_{l\in L} [asgn_{o,l,t} \cdot (1 - LOCAL_{o,l})] \cdot LFORL_{o,l}^{\$} \cdot \omega_3 && (3.12)\\
+ & \sum_{t\in T_{>0}} \sum_{o\in O} \sum_{l\in L} aaux_{o,l,t} \cdot CONT^{\$} \cdot \omega_4 && (3.13)\\
+ & \sum_{t\in T} \sum_{p\in P} \sum_{l\in L} upgr_{l,p,t} \cdot UPG_{l,p}^{\$} \cdot \omega_5 \cdot 2^{|T|-t} + newb_{l,t} \cdot BLD^{\$} && (3.14)
\end{aligned}$$

And for the SMT formulation:

$$\begin{aligned}
\min & \sum_{t\in T} \sum_{l\in L} uaux_{l,t} \cdot UTIL_l^{\$} \cdot \omega_1 && (3.10)\\
+ & \sum_{o\in O} \sum_{l\in L} ITE(buyrel_{o,l}, RELEASE_o^{\$} \cdot \omega_2, 0) && (3.10)\\
+ & \sum_{t\in T} \sum_{o\in O} \sum_{l\in L} ITE(asgn_{o,l,t} \land \neg LOCAL_{o,l} \cdot \omega_3, LFORL_{o,l}^{\$}, 0) && (3.12)\\
+ & \sum_{t\in T_{>0}} \sum_{o\in O} \sum_{l\in L} ITE(aaux_{o,l,t}, CONT^{\$} \cdot \omega_4, 0) && (3.13)\\
+ & \sum_{t\in T} \sum_{p\in P} \sum_{l\in L} ITE(upgr_{l,p,t}, UPG \cdot \omega_5 \cdot 2^{|T|-t} + newb_{l,t} \cdot BLD^{\$}, 0) && (3.14)
\end{aligned}$$

With the added constraints as we have just discussed.

3.2 The Set of Constraints: Reduced Problem

In order for our objective function to give us any meaningful results, we need to define the solution space, onto which it is supposed to be applied. This is done by a set of constraints, which restrict the solution space to the space of consistent and realistic solutions. In this section, we want to capture the planning problem while ignoring upgrades, line configurations and the continuity of assignments to a line for the moment.

There are two constraints that may come to mind right away, as they can be seen as the most natural constraints of the problem. First, the amount of parts that each order specifies should be completely produced. Secondly, to do so, a line may not produce more parts than physically possible in each time period.
The first constraint can be modelled as follows: In each time period and for each order, ensure that the amount produced over all lines is equal to the amount that is supposed to be produced during this time period. The mathematical formulation is identical for both the SMT and the MIP-formulation:

$$\sum_{l \in L} prod_{o,l,t} = SIZE_{o,t} \quad \forall o \in O, t \in T \tag{3.15}$$

For our second constraint, the amount produced over all orders on a single line in every time period is to be smaller or equal to the capacity that this line is able to provide. Again, the formulation is the same for both SMT and MIP:

$$\sum_{o \in O} prod_{o,l,t} \leq CAPM_{l,t} \quad \forall l \in L, t \in T \tag{3.1}$$

As each product that is produced at the production sites of the Robert Bosch GmbH can vary in its required features, not every line is able to produce the parts of every order. For this purpose, we are given the information which line is able to produce which order initially. In order to restrict our possible assignments of orders to lines, we choose the following constraint - first formulated for a MIP and then for SMT:

$$asgn_{o,l,t} \leq POSS_{o,l} \quad \forall o \in O, l \in L, t \in T \tag{3.2}$$
$$asgn_{o,l,t} \rightarrow POSS_{o,l} \quad \forall o \in O, l \in L, t \in T \tag{3.2}$$

The binary constant $POSS_{o,l}$ only reflects the possible assignments at the very start of production and is thus only suited for our reduced model. We will discuss the changes that are needed for the complete model in the respective sections.
The next point that we have to care for is that of already given assignments of orders to lines or production sites. As the production process for which we are trying to create a plan for the next years is already in action today, we cannot simply disregard the current established plan. For this purpose, we are given a constant $INITLINE_{o,l}$ for every order o and line l, stating whether in time period 0, i.e., the start of our plan, an order is already assigned to a line.

$$INITLINE_{o,l} \leq asgn_{o,l,0} \quad \forall o \in O, l \in L \tag{3.17}$$
$$INITLINE_{o,l} \rightarrow asgn_{o,l,0} \quad \forall o \in O, l \in L \tag{3.17}$$

Similarly, an order may not be assigned to a specific line, but to a production site. We are given this information in the form of the constant $INITSITE_{o,s}$.

$$INITSITE_{o,s} \leq atsite_{o,s,0} \quad \forall o \in O, s \in S \tag{3.25}$$
$$INITSITE_{o,s} \rightarrow atsite_{o,s,0} \quad \forall o \in O, s \in S \tag{3.25}$$

Finally, an order may be needed to be produced during a certain range of time periods or even exclusively at a certain site. For this purpose, we are given the constants $FIXSITES_{o,s,t}$ which indicate whether an order o is fixated to a site s at time period t. The corresponding constraint is the following:

$$FIXSITES_{o,s,t} \leq atsite_{o,s,t} \quad \forall o \in O, s \in S, t \in T \tag{3.26}$$

$$FIXSITES_{o,s,t} \rightarrow atsite_{o,s,t} \quad \forall o \in O, s \in S, t \in T \tag{3.26}$$

One can see that the variable $atsite_{o,s,t}$ states whether an order o is produced at site s in time period t. In order for this initial assignment to take effect, we need to couple the value of this variable with another one of our decision variables, i.e. the variable $asgn_{o,l,t}$ which encodes whether an order o is assigned to a line l during some time period t. We want two directions of linking these variables in each time period t:

- If the value of $asgn_{o,l,t}$ becomes true, order o is produced at the site s of line l.
- If the value of $atsite_{o,s,t}$ becomes true, order o is assigned to one of the lines of site s.

We formalize these two directions as follows as a MIP formulation:

$$atsite_{o,s,t} \geq asgn_{o,l,t} \cdot LINKED_{l,s} \quad \forall o \in O, l \in L, s \in S, t \in T \tag{3.23}$$

$$atsite_{o,s,t} \leq \sum_{l \in L} asgn_{o,l,t} \cdot LINKED_{l,s} \quad \forall o \in O, s \in S, t \in T \tag{3.24}$$

And similarly for the SMT formulation:

$$(asgn_{o,l,t} \land LINKED_{l,s}) \rightarrow atsite_{o,s,t} \quad \forall o \in O, l \in L, s \in S, t \in T \tag{3.23}$$

$$atsite_{o,s,t} \rightarrow \bigvee_{l \in L} (asgn_{o,l,t} \land LINKED_{l,s}) \quad \forall o \in O, s \in S, t \in T \tag{3.24}$$

Next, we encode the fact that for producing certain orders on certain lines, customer releases may have to be bought. While our objective function already models the concrete costs of buying customer releases, we do not yet restrict the variables $buyrel_{o,l}$ that encode whether a customer release has to be bought, so they may all be set to 0 as of now. For this purpose, similar to our $POSS_{o,l}$ constant, we use a constant $REL_{o,l}$ that indicates if for producing an order o on line l a release is already present.

$$buyrel_{o,l} \geq (1 - REL_{o,l}) \cdot asgn_{o,l,t} \quad \forall o \in O, l \in L, t \in T \tag{3.22}$$

$$(\neg REL_{o,l} \land asgn_{o,l,t}) \rightarrow buyrel_{o,l} \quad \forall o \in O, l \in L, t \in T \tag{3.22}$$

To conclude the needed constraints for the mentioned reduced mathematical problem, we need to link a few more variables with one another. If our variable $prod_{o,l,t}$ indicates that some amount of parts from order o are to be produced on line l in time period t, then the order is also be assigned to this line at the specified time period.

$$prod_{o,l,t} \leq asgn_{o,l,t} \cdot CAPM_{l,t} \quad \forall o \in O, l \in L, t \in T \tag{3.19}$$

$$\neg asgn_{o,l,t} \rightarrow (prod_{o,l,t} = 0) \quad \forall o \in O, l \in L, t \in T \tag{3.19}$$

Lastly, when an order is assigned to a line, then this line becomes active for this time period. The other direction should hold as well, i.e. we are only allowed to activate lines if any order is assigned to them.

$$asgn_{o,l,t} \leq actv_{l,t} \quad \forall o \in O, l \in L, t \in T \tag{3.18}$$

$$asgn_{o,l,t} \rightarrow actv_{l,t} \quad \forall o \in O, l \in L, t \in T \tag{3.18}$$

$$actv_{l,t} \leq \sum_{o \in O} asgn_{o,l,t} \quad \forall l \in L, t \in T \tag{3.20}$$

$$actv_{l,t} \rightarrow \bigvee_{o \in O} asgn_{o,l,t} \quad \forall l \in L, t \in T \tag{3.20}$$

Note that by these couplings, the variables $actv_{l,t}$ and $prod_{o,l,t}$ are transitively coupled as well.

3.3 The Set of Constraints: Complete Problem

Now that we have discussed the constraints needed for the basic model, we focus on the constraints needed for the complete model, i.e. everything needed for allowing the change of line types in form of upgrades and continuity of assignments.

If we allow line types to change over time, lines may in consequence gain or lose the ability to produce certain orders. This means that the constraint 3.2 is not sufficient anymore as it can only make statements over the initial time period 0. To replace it in our complete model, we make use of another binary constant named $ASGN_{o,p}$ which states whether an order o can be produced on an arbitrary line of configuration p. We can thus replace our constant $POSS_{o,l}$ and reformulate our constraint using the information which type a line is currently configured to:

$$asgn_{o,l,t} \leq \sum_{p \in P} ASGN_{o,p} \cdot islt_{l,p,t} \quad \forall o \in O, l \in L, t \in T \tag{3.21}$$

$$asgn_{o,l,t} \rightarrow \bigvee_{p \in P} (ASGN_{o,p} \wedge islt_{l,p,t}) \quad \forall o \in O, l \in L, t \in T \tag{3.21}$$

A special requirement given by the Robert Bosch GmbH was that of line utilization, as introduced when discussing the objective function in section 3.1. While we already attended to the program trying to reach a certain amount of utilization, there is a second aspect that goes along with this first one. If a line is not used during a whole time period, it is supposed to be deactivated and deconstructed. The idea is to identify lines that are not necessary for production of all orders such that resources of the company can be saved.
There is one special case that we have to be aware of: There may be lines that may still be in construction when we calculate our plan. These constructions are not explicitly given and can only be identified by the fact that a line may not have any capacities during some of the first time periods, although already having the line configuration of a regular line. From these capacities, we can construct constants $FT_{l,t}$ that become true at exactly the first time period after the initial time period 0 at which the capacities of a regular line rise from 0 to some nonnegative, nonzero amount. Over all time periods, this constant thus evaluates at most once to $True$ for every line. Apart from this special case, this constraint is rather easy to model. We simply let the value of our variable $actv_{l,t}$ depend on the value of the previous time period.

$$actv_{l,t} \leq actv_{l,t-1} + FT_{l,t} \quad \forall l \in L, t \in T_{>0} \tag{3.35}$$

$$actv_{l,t} \rightarrow actv_{l,t-1} \vee FT_{l,t} \quad \forall l \in L, t \in T_{>0} \tag{3.35}$$

We will shortly see that the given formulation would be sufficient in a scenario in which lines are assumed to be either activated for production or deactivated for upcoming deconstruction.

With the possibility of upgrading lines at hand, we have to regard a third scenario: A line may be currently deactivated because it is not constructed yet. In this case, being able to activate the line after building it is desirable. We will revisit this constraint later in the chapter to see which possibilities we have to encode this property.

As with the releases that can be assumed to having been bought before the calculated plan is put into action, we can assume some upgrades to be completed before production starts. The following properties regarding upgrades now additionally need to be encoded:

- A line may not be upgradeable to every possible configuration.
- There may be more than one upgrade for every line.
- After being built, a line should be able to become active.
- The current linetype has be determined based on the initial linetype and possibly multiple upgrades that have happened in previous time periods.
- The construction times of a line have to be honored.
- The capacities of newly built lines are ramping up over time and are dependent on the time since construction.

We will discuss two ways of encoding these constraints and which advantages and disadvantages they bear. Their main difference stems from how we decide to encode the property of upgrading a line in a variable. The first variant uses a binary variable $upgr_{l,p_1,p_2,t}$ in order to keep track of which line l in which time period t was upgraded from which line configuration p_1 to which other configuration p_2. We will call this variant *compressed* as our upgrade variable holds most of the relevant information for the constraints it is involved in.
Our second variant of this variable is almost identical to the first one, with the difference of not encoding the line configuration a line was upgraded *from*. That is, the upgrade variable of our second variant is $upgr_{l,p,t}$ with p only encoding to which configuration a line was upgraded. We will call this second variant *distributed* as most of the relevant information for the constraints is distributed among the *upgr*-variable and other auxiliary variables. In both cases, the variable *upgr* evaluates to $True$ if the line construction is *completed* in the current time period, i.e. the line is ready to produce in the current time period. We will discuss the advantages and disadvantages of both versions in section 3.3.3.

There is one constraint that both variants share, as it is independent from the *upgr* variable. Any line l has exactly one type of configuration p during every time period. For lines that are not constructed yet, this is the “empty” line type P_0:

$$\sum_{p \in P} islt_{l,p,t} = 1 \quad \forall l \in L, t \in T \tag{3.36}$$

$$\underset{p \in P}{ExactlyOne}(islt_{l,p,t}) \quad \forall l \in L, t \in T \tag{3.36}$$

We further note that due to possible constructions of new lines, constraint 3.19 has to be adjusted for the MIP-version, as the $CAPM_{l,t}$ constants are inherently 0 for lines that are not yet constructed. This will be further discussed in the following subsections.

$$prod_{o,l,t} \leq asgn_{o,l,t} \cdot (\sum_{i=0}^{|T|-1} CAPNM_{l,i} + CAPM_{l,t}) \quad \forall o \in O, l \in L, t \in T \tag{3.19}$$

3.3.1 Compressed Variant

First, not every line can be upgraded to every configuration of the set P. In order to limit the search space to the set of possible upgrades, we make use of the constants $UPGRABLE_{l,p}$ which state for every line l and configuration p whether this specific upgrade is possible or not. Thus, we can combine this constant with our *upgr*-variables very easily:

$$upgr_{l,p_1,p_2,t} \leq UPGRABLE_{l,p_2} \quad \forall l \in L, p_1, p_2 \in P, t \in T \tag{3.3}$$

$$upgr_{l,p_1,p_2,t} \rightarrow UPGRABLE_{l,p_2} \quad \forall l \in L, p_1, p_2 \in P, t \in T \tag{3.3}$$

As we determine which assignment of orders to lines is possible based on the current configuration of a line, we need to couple the variables *islt* and *upgr* with one another. First, we look at the case that is very straight-forward: If a line is upgraded to a certain type p_2, its configuration should change accordingly:

$$\sum_{p_1 \in P} upgr_{l,p_1,p_2,t} \leq islt_{l,p_2,t} \quad \forall l \in L, p_2 \in P, t \in T \tag{3.4}$$

$$\bigvee_{p_1 \in P} upgr_{l,p_1,p_2,t} \rightarrow islt_{l,p_2,t} \quad \forall l \in L, p_2 \in P, t \in T \tag{3.4}$$

We are not done with the coupling of the variables yet. While the previous constraint correctly sets the linetype once an upgrade is completed, we still need to correctly determine the type of a line in between updates. For this, we distinguish two cases.
A line l is of type p if:

- This was its initial configuration and until the current time period, no upgrade was made on this line.
- This was not its initial configuration, it was however upgraded to it and since then, no second upgrade was made away from the current type.

We formalize these two statements for the MIP:

$$islt_{l,p,t} \leq 1 - \sum_{i=0}^{t} upgr_{l,p,p_2,i} \quad \forall l \in L, p, p_2 \in P, t \in T \wedge ILT_{l,p} \tag{3.5}$$

$$islt_{l,p,t} \leq \sum_{i=0}^{t} \Big(\sum_{p_1 \in P} upgr_{l,p_1,p,i} - \sum_{p_2 \in P} upgr_{l,p,p_2,i} \Big) \quad \forall l \in L, p \in P, t \in T \wedge \neg ILT_{l,p} \tag{3.6}$$

And as an SMT formulation:

$$islt_{l,p,t} \rightarrow \neg \bigvee_{i=0}^{t} upgr_{l,p,p_2,i} \quad \forall l \in L, p_1, p_2 \in P, t \in T \wedge ILT_{l,p} \tag{3.5}$$

$$islt_{l,p,t} \rightarrow \bigvee_{i=0}^{t} \Big(\bigvee_{p_1 \in P} upgr_{l,p_1,p,i} \wedge \neg \bigvee_{p_2 \in P} upgr_{l,p,p_2,i} \Big) \quad \forall l \in L, p \in P, t \in T \wedge \neg ILT_{l,p} \tag{3.6}$$

Looking at constraint 3.5 of the MIP formulation, we see that we run into problems if a line is upgraded away from its initial line type to some other configuration, then back to its initial line type and then again upgraded. In this scenario, the sum of variables in the constraint may become larger or equal 2, resulting in our whole formulation to be infeasible, as we would force a binary variable to evaluate to a value smaller than 0. As the situation that we just described is not desirable but also not quite realistic, we choose to restrict lines to be of a fixed configuration p only once over the whole planning horizon - in this variant - by the following constraint:

$$\sum_{t \in T} \sum_{p_1 \in P} upgr_{l,p_1,p,t} \leq 1 \quad \forall l \in L, p \in Pt \in T \tag{3.7}$$

$$\underset{t \in T, p_1 \in P}{AtMostOne}(upgr_{l,p_1,p,t}) \quad \forall l \in L, p \in Pt \in T \tag{3.7}$$

Next, we do have to restrict our solver from ignoring the construction time of a production line. While upgrades can be seen as instantaneous, newly constructing a line takes BT time periods.

$$1 - upgr_{l,P_0,p,t} \geq \sum_{i=max\{t-BT,0\}} \sum_{p_2 \in P} upgr_{l,P_0,p_2,i} \quad \forall l \in L, p \in P, t \in T \tag{3.8}$$

$$\bigvee_{i=max\{t-BT,0\}} \bigvee_{p_2 \in P} upgr_{l,P_0,p_2,i} \rightarrow \neg upgr_{l,P_0,p,t} \quad \forall l \in L, p \in P, t \in T \tag{3.8}$$

Finally, we revisit the constraint of line continuity. Given our compressed variant, we now can easily state that either a line has been active in the previous time period in order to still remain active in the current time period, or a construction - an upgrade from the “empty” line type - is completed in the current time period or a line is only able to start production for the first time in the current time period:

$$actv_{l,t} \leq actv_{l,t-1} + \sum_{p_2 \in P} upgr_{l,P_0,p_2,max\{t-BT_l,0\}} + FT_{l,t} \quad \forall l \in L, t \in T_{>0} \tag{3.35}$$

$$actv_{l,t} \rightarrow actv_{l,t-1} \vee \bigvee_{p_2 \in P} upgr_{l,P_0,p_2,max\{t-BT_l,0\}} \vee FT_{l,t} \quad \forall l \in L, t \in T_{>0} \tag{3.35}$$

Note that “upgrading” a line to P_0 is not possible as we explicity exclude this option while acquiring our data.
When newly constructing a line, it is not able to start production at full capacity right away. Rather, its capacity is slowly ramping up until it reaches its maximum. For this purpose, we are given constants $CAPNM_{l,t}$ for every line which encode this rampup in the following way: $CAPNM_{l,0}$ encodes the maximal capacities line l has in the time period in which the construction of the line is completed. Accordingly, $CAPNM_{l,1}$ is the capacity of the line in the first time period after construction and so on.
We can assume the following: For every line that is initially already constructed, the values of $CAPNM_{l,t}$ will always be 0 and the values of $CAPM_{l,t}$ will generally be unequal to 0. For lines that are initially not constructed, the opposite is the case: The values of $CAPM_{l,t}$ will always be 0 and the values of $CAPNM_{l,t}$ will generally be unequal to 0. This observation makes a reformulation of our initial line capacity constraint 3.1 rather easy:

$$\sum_{o \in O} prod_{o,l,t} \leq \sum_{i=0}^{t} (CAPNM_{l,i} \cdot (\sum_{p \in P} upgr_{l,P_0,p,i})) + CAPM_{l,t} \quad \forall l \in L, t \in T \tag{3.9}$$

3.3.2 Distributed Variant

The distributed variant of our constraints does not keep track of the linetype a line was upgraded from, as we have mentioned before. This means that at each point in which we have previously used the explicit linetype that we upgraded from, we now have to find a workaround in order to

obtain the same information.
Upgrading a line is even more straightforward than in our other variant, as we can simply couple each pair of $upgr_{l,p,t}$ variables and $UPGRABLE_{l,p}$ constants:

$$upgr_{l,p,t} \leq UPGRABLE_{l,p} \quad \forall l \in L, p \in P, t \in T \tag{3.31}$$
$$upgr_{l,p,t} \rightarrow UPGRABLE_{l,p} \quad \forall l \in L, p \in P, t \in T \tag{3.31}$$

Coupling our *islt* variables with our $upgr_{l,p,t}$ variables is just as simple:

$$upgr_{l,p,t} \leq islt_{l,p,t} \quad \forall l \in L, p \in P, t \in T \tag{3.37}$$
$$upgr_{l,p,t} \rightarrow islt_{l,p,t} \quad \forall l \in L, p \in P, t \in T \tag{3.37}$$

The coupling in the other direction becomes a bit more involved. In our compressed variant, we could make statements about a line “losing” its configuration p by stating that an upgrade with p as its basis was made. Obviously, this is no longer possible, so we introduce a new auxiliary variable $lwu_{l,t}$, which simply encodes whether a given line l was upgraded any number of times up to the current time period t or if no single upgrade was made to it. We propose the following three constraints in order to completely determine the linetype of every line:

$$islt_{l,p,t} \leq 1 - lwu_{l,t} \quad \forall l \in L, p \in P, t \in T \wedge ILT_{l,p} \tag{3.32}$$
$$islt_{l,p,t} \leq islt_{l,p,t-1} + upgr_{l,p,t} \quad \forall l \in L, p \in P, t \in T \wedge \neg ILT_{l,p} \tag{3.33}$$
$$islt_{l,p,0} \leq upgr_{l,p,0} \quad \forall l \in L, p \in P \wedge \neg ILT_{l,p} \tag{3.34}$$

$$islt_{l,p,t} \rightarrow \neg lwu_{l,t} \quad \forall l \in L, p \in P, t \in T \wedge ILT_{l,p} \tag{3.32}$$
$$islt_{l,p,t} \rightarrow islt_{l,p,t-1} \vee upgr_{l,p,t} \quad \forall l \in L, p \in P, t \in T \wedge \neg ILT_{l,p} \tag{3.33}$$
$$islt_{l,p,0} \rightarrow upgr_{l,p,0} \quad \forall l \in L, p \in P \wedge \neg ILT_{l,p} \tag{3.34}$$

We want to shortly discuss why constraint 3.33 is sufficient in order to correctly force a consistent and correct configuration for every line over every time period beyond the first one. Using constraints 3.32 and 3.34, we force every single line to have a unique configuration in time period 0: For every line l there is exactly one configuration p such that $ILT_{l,p}$ is *True*, i.e. every line has a unique initial line type. It may however still be the case that this initial line type is no longer valid in time period 0 because an update was completed in this time period. For this purpose, 3.34, together with constraint 3.37 correctly forces the variable to its new type.
This is the reason why in constraint 3.33, we can always state that either a line still has the type of its previous time period or that an upgrade was completed in the current time period, which allows a line to be of another configuration.
As we have used the newly introduced $lwu_{l,t}$ variable, we need to assure that its values are consistent with the rest of our formulation. We thus couple the variable with our $upgr_{l,p,t}$ variable in two directions. This is very simple in SMT:

$$lwu_{l,t} \leftrightarrow \bigvee_{p \in P} \bigvee_{i=0}^{t} upgr_{l,p,i} \quad \forall l \in L, t \in T \tag{3.38}$$

For an MIP formulation however, a single type of constraint is not sufficient. Our first type of constraints states that the value of $lwu_{l,t}$ can only become $True$ if some upgrade on the given line l was made up to and including the time period t:

$$lwu_{l,t} \leq \sum_{p \in P} \sum_{i=0}^{t} upgr_{l,p,i} \quad \forall l \in L, t \in T \tag{3.38}$$

For the other direction, we state that when an upgrade to the line l is made, the value of $lwu_{l,t}$ should be set to $True$ accordingly:

$$lwu_{l,t} \cdot min(t+1, |T|) \geq \sum_{p \in P} \sum_{i=0}^{t} upgr_{l,p,i} \forall l \in L, t \in T$$

This constraint is correct in principle but is rather "loose" for the MIP model, as it is utilizing a "big M" coupling. We can reformulate this constraint using two other constraints which result in a "tighter" formulation:

$$lwu_{l,t} \geq lwu_{l,t-1} \quad \forall l \in L, t \in T_{>0} \tag{3.40}$$
$$upgr_{l,p,t} \leq lwu_{l,t} \quad \forall l \in L, p \in P, t \in T \tag{3.39}$$

We turn to our problem of continuously using lines or deconstructing them. This is the other case in which we cannot use the formulation of our compressed variant, as we cannot directly state that a line was upgraded from the "empty" configuration P_0 in our MIP. Again, we need to use an auxiliary variable $newb_{l,t}$ which states whether a line l is upgraded from P_0 to some other configuration p in time period t:

$$actv_{l,t} \leq actv_{l,t-1} + newb_{l,t} + FT_{l,t} \quad \forall l \in L, t \in T_{>0} \tag{3.35}$$
$$actv_{l,t} \rightarrow actv_{l,t-1} \vee newb_{l,t} \vee FT_{l,t} \quad \forall l \in L, t \in T_{>0} \tag{3.35}$$

Again, note that "upgrading" a line to P_0 is not possible as we explicity exclude this possibility while acquiring our data.
As in our compressed variant, we want to hinder our solver from ignoring the time that it takes to newly construct a line. For this, we utilize our $newb_{l,t}$ variables:

$$1 - newb_{l,t} \geq \sum_{p \in P} \sum_{i=max\{t-BT,0\}}^{t-1} upgr_{l,p,i} \quad \forall l \in L, t \in T \tag{3.43}$$
$$\bigvee_{p \in P} \bigvee_{i=max\{t-BT,0\}}^{t-1} upgr_{l,p,i} \rightarrow \neg newb_{l,t} \quad \forall l \in L, t \in T \tag{3.43}$$

As we have already mentioned when constructing constraint 3.9 of our compressed variant, the line capacities, given by $CAPNM_{l,t}$ are reduced during the first time periods after construction. We can model this constraint very similarly, utilizing our $newb_{l,t}$ variables instead of our $upgr_{l,p_1,p_2,t}$ variables:

$$\sum_{o \in O} prod_{o,l,t} \leq \sum_{i=0}^{t} (CAPNM_{l,i} \cdot newb_{l,t-i}) + CAPM_{l,t} \quad \forall l \in L, t \in T \tag{3.16}$$

Lastly, we need to couple the value of this newly introduced auxiliary variable $newb_{l,t}$ with the rest of our variables in order to let it only obtain correct and consistent valuations:

$$2 \cdot newb_{l,t} \leq \sum_{p \in P} upgr_{l,p,t} + (1 - lwu_{l,t-1}) \cdot ILT_{l,P_0} \quad \forall l \in L, t \in T_{>0} \tag{3.41}$$
$$2 \cdot newb_{l,0} \leq \sum_{p \in P} upgr_{l,p,0} + ILT_{l,P_0} \quad \forall l \in L \tag{3.42}$$

$$newb_{l,t} \leftrightarrow \bigvee_{p \in P} upgr_{l,p,t} \wedge \neg lwu_{l,t-1} \wedge ILT_{l,P_0} \quad \forall l \in L, t \in T_{>0} \tag{3.41}$$

$$newb_{l,0} \leftrightarrow \bigvee_{p \in P} upgr_{l,p,0} \wedge ILT_{l,P_0} \quad \forall l \in L \tag{3.42}$$

Note that since the solver does not gain any advantage by setting the value of $newb_{l,t}$ to $False$ - it does not allow him to activate lines or make use of the capacities of the constructed lines -, it is sufficient that our MIP-constraints limit the variables in the upper direction.

3.3.3 A Comparison Between the Two Variants

In this section, we want to discuss the reasoning behind introducing two different formulations of the same problem and how they compare.
Through the course of constructing the constraints of the problem at hand, we first formulated the compressed variant, as it seemed like a very natural way to calculate the upgrade costs in our objective function, which would then precisely reflect the costs of every single feature that was added.
Formulating the constraints in the way that we have documented them in this chapter reveals however, that we need to make a compromise in order to correctly determine the current configuration of a line - i.e. $islt_{l,p,t}$ - in constraint 3.5. This compromise is that of the constraint 3.7, which states that each line can only be configured to the same configuration only once over all time periods. This is necessary as otherwise, the sums used in the aforementioned constraints could result in values bigger than one and in consequence, an otherwise satisfiable problem would become unsatisfiable.
For practical usage, this does not really restrict the plan, as upgrading from a certain line type p_1 to some other line type p_2 and then later back to p_1 does not make sense economically, as each reconfiguration is costly. Especially, there is no general use for downgrading a line except for possibly very specific reasons that go beyond the scope of this formulation.
There is, however a much more pressing issue with this formulation in that the number of all $upgr_{l,p_1,p_2,t}$ variables scales quadratically with the size of the set P. As performance, especially with regards to memory consumption became an issue, we formulated the distributed variant of our problem as an alternative which we also use in our implementation.
While the variable space of the $upgr_{l,p,t}$ variables scales only linearly in the size of P, formulating equivalent constraints to those of the compressed version becomes a new challenge as we no longer know from which type an upgrade is made. This results in the introduction of the additional variables $lwu_{l,t}$ and $newb_{l,t}$ of which the first states whether a line was already upgraded at least once up to and including the current time period, and the latter variable indicating whether a line is newly constructed in the current time period.
While this formulation no longer relies on the assumption that each linetype can only be taken once for every line, formulating an equivalent for the objective function proves to be challenging. When using our reformulated upgrade variable, we are only able to state what the costs of upgrading a line to a certain configuration is, measured relative to its initial configuration, as we lack the information from which configuration the upgrade is made from. In order to see how this proves to be a problem and to show how our solution to this problem works, we take a look at the following small example.

Example 3.3.1. Assume a line l with some initial configuration of features F of which we assume that it does not include the features f_1 and f_2. In order to produce an order that is assigned to it in time period 1, it needs to add the feature f_1. For another order that is assigned to this line in time period 3, the features f_1 and f_2 are required, such that the final configuration of the line is f_1 and f_2 (additional to the features given by the initial configuration F).

In this example, our compressed model would have the choice between first adding the feature f_1 to the line l only to later add the feature f_2 and the option of adding both features at once, as the costs of both upgrades are the same, since we do not pay for the cost of upgrading itself. I.e., if $c(\{features\})$ is a function that projects adding a set of features to the cost of adding these features, $c(\{f_1\}) + c(\{f_2\}) = c(\{f_1, f_2\})$ holds for our compressed variant.
Our distributed variant, however, would always choose the option to directly add both features at once, since the upgrade costs are always compared to the initial configuration of the concrete line, as it is the only common point of reference. Thus, it would "pay" twice for feature f_1, i.e. $c(\{f_1\}) + c(\{f_1, f_2\}) > c(\{f_1, f_2\})$. This is undesirable for the Robert Bosch GmbH, since upgrades are supposed to be made only when they become necessary, i.e. a line should only gain features that are necessary for production in this time period. For this purpose, we add an exponential backoff to the objective function, namely the term $2^{(|T|-t)}$. This means that over all possible upgrades in all time periods, there is always a solution in which buying the same upgrade as late as possible is cheapest. In both the distributed and the compressed variant, the solvers will thus try to buy the cheapest upgrade that satisfies the minimal requirements for production local to the current time period t. Thus, especially our distributed variant will now work as desired regarding upgrades. A side effect of this multiplication is that the cost of upgrades now dominates the objective function, thus if we weight all parts of the objective equally, we will find an optimal solution regarding minimal upgrade costs.

3.4 A Complete Mathematical Formulation

Based on the descriptions of the previous paragraphs we are able to construct the following mathematical formulation for modelling the given problem:

$$\min \sum_{t\in T}\sum_{l\in L} uaux_{l,t} \cdot UTIL^{\$}_{l} \cdot \omega_1 \tag{3.10}$$

$$+\sum_{o\in O}\sum_{l\in L} buyrel_{o,l} \cdot RELEASE^{\$}_{o} \cdot \omega_2 \tag{3.11}$$

$$+\sum_{t\in T}\sum_{o\in O}\sum_{l\in L} [asgn_{o,l,t} \cdot (1 - LOCAL_{o,l})] \cdot LFORL^{\$}_{o,l} \cdot \omega_3 \tag{3.12}$$

$$+\sum_{t\in T_{>0}}\sum_{o\in O}\sum_{l\in L} aaux_{o,l,t} \cdot CONT^{\$} \cdot \omega_4 \tag{3.13}$$

$$+\sum_{t\in T}\sum_{p\in P}\sum_{l\in L} upgr_{l,p,t} \cdot UPG^{\$}_{l,p} \cdot \omega_5 \cdot 2^{|T|-t} + newb_{l,t} \cdot BLD^{\$} \tag{3.14}$$

$$\text{s.t.} \sum_{l\in L} prod_{o,l,t} = SIZE_{o,t} \qquad \forall o \in O, t \in T \tag{3.15}$$

$$\sum_{o\in O} prod_{o,l,t} \leq \sum_{i=0}^{t} (CAPNM_{l,i} \cdot newb_{l,t-i}) + CAPM_{l,t} \qquad \forall l \in L, t \in T \tag{3.16}$$

$$INITLINE_{o,l} \leq asgn_{o,l,0} \qquad \forall o \in O, l \in L \tag{3.17}$$

$$asgn_{o,l,t} \leq actv_{l,t} \qquad \forall o \in O, l \in L, t \in T \tag{3.18}$$

$$prod_{o,l,t} \leq asgn_{o,l,t} \cdot (\sum_{i=0}^{|T|-1} CAPNM_{l,i} + CAPM_{l,t}) \qquad \forall o \in O, l \in L, t \in T \tag{3.19}$$

$$actv_{l,t} \leq \sum_{o\in O} asgn_{o,l,t} \qquad \forall l \in L, t \in T \tag{3.20}$$

$$asgn_{o,l,t} \leq \sum_{p \in P} ASGN_{o,p} \cdot islt_{l,p,t} \qquad \forall o \in O, l \in L, t \in T \tag{3.21}$$

$$buyrel_{o,l} \geq (1 - REL_{o,l}) \cdot asgn_{o,l,t} \qquad \forall o \in O, l \in L, t \in T \tag{3.22}$$

$$atsite_{o,s,t} \geq asgn_{o,l,t} \cdot LINKED_{l,s} \qquad \forall o \in O, l \in L, s \in S, t \in T \tag{3.23}$$

$$atsite_{o,s,t} \leq \sum_{l \in L} asgn_{o,l,t} \cdot LINKED_{l,s} \qquad \forall o \in O, s \in S, t \in T \tag{3.24}$$

$$INITSITE_{o,s} \leq atsite_{o,s,0} \qquad \forall o \in O, s \in S \tag{3.25}$$

$$FIXSITES_{o,s,t} \leq atsite_{o,s,t} \qquad \forall o \in O, s \in S, t \in T \tag{3.26}$$

$$uaux_{l,t} \geq TUTIL_{l,t} \cdot actv_{l,t} - \frac{\Sigma_{o \in O} prod_{o,l,t}}{CAPM_{l,t} + CAPNM_{l,t} + \varepsilon} \qquad \forall l \in L, t \in T \tag{3.27}$$

$$uaux_{l,t} \geq \frac{\Sigma_{o \in O} prod_{o,l,t}}{CAPM_{l,t} + CAPNM_{l,t} + \varepsilon} - TUTIL_{l,t} \cdot actv_{l,t} \qquad \forall l \in L, t \in T \tag{3.28}$$

$$aaux_{o,l,t} \geq asgn_{o,l,t-1} - asgn_{o,l,t} \qquad \forall o \in O, l \in L, t \in T_{>0} \tag{3.29}$$

$$aaux_{o,l,t} \geq asgn_{o,l,t} - asgn_{o,l,t-1} \qquad \forall o \in O, l \in L, t \in T_{>0} \tag{3.30}$$

$$upgr_{l,p,t} \leq UPGRABLE_{l,p} \qquad \forall l \in L, p \in P, t \in T \tag{3.31}$$

$$islt_{l,p,t} \leq 1 - lwu_{l,t} \qquad \forall l \in L, p \in P, t \in T \land ILT_{l,p} \tag{3.32}$$

$$islt_{l,p,t} \leq islt_{l,p,t-1} + upgr_{l,p,t} \qquad \forall l \in L, p \in P, t \in T \land \neg ILT_{l,p} \tag{3.33}$$

$$islt_{l,p,0} \leq upgr_{l,p,0} \qquad \forall l \in L, p \in P \land \neg ILT_{l,p} \tag{3.34}$$

$$actv_{l,t} \leq actv_{l,t-1} + newb_{l,t} + FT_{l,t} \qquad \forall l \in L, t \in T_{>0} \tag{3.35}$$

$$\sum_{p \in P} islt_{l,p,t} = 1 \qquad \forall l \in L, t \in T \tag{3.36}$$

$$upgr_{l,p,t} \leq islt_{l,p,t} \qquad \forall l \in L, p \in P, t \in T \tag{3.37}$$

$$lwu_{l,t} \leq \sum_{p \in P} \sum_{i=0}^{t} upgr_{l,p,i} \qquad \forall l \in L, t \in T \tag{3.38}$$

$$upgr_{l,p,t} \leq lwu_{l,t} \qquad \forall l \in L, p \in P, t \in T \tag{3.39}$$

$$lwu_{l,t} \geq lwu_{l,t-1} \qquad \forall l \in L, t \in T_{>0} \tag{3.40}$$

$$2 \cdot newb_{l,t} \leq \sum_{p \in P} upgr_{l,p,t} + (1 - lwu_{l,t-1}) \cdot ILT_{l,P_0} \qquad \forall l \in L, t \in T_{>0} \tag{3.41}$$

$$2 \cdot newb_{l,0} \leq \sum_{p \in P} upgr_{l,p,0} + ILT_{l,P_0} \qquad \forall l \in L \tag{3.42}$$

$$1 - newb_{l,t} \geq \sum_{p \in P} \sum_{i=max\{t-BT,0\}}^{t-1} upgr_{l,p,i} \qquad \forall l \in L, t \in T \tag{3.43}$$

$$aaux_{o,l,t} \in \mathbb{B}_+ \qquad \forall o \in O, l \in L, t \in T_{>0}$$

$$uaux_{l,t} \in \mathbb{R}_+ \qquad \forall l \in L, t \in T$$

$$prod_{o,l,t} \in \mathbb{R}_+ \qquad \forall o \in O, l \in L, t \in T$$

$$asgn_{o,l,t} \in \mathbb{B} \qquad \forall o \in O, l \in L, t \in T$$

$$atsite_{o,s,t} \in \mathbb{B} \qquad \forall o \in O, s \in S, t \in T$$

$$actv_{l,t} \in \mathbb{B} \qquad \forall l \in L, t \in T$$

$$buyrel_{o,l} \in \mathbb{B} \qquad \forall o \in O, l \in L$$

$$upgr_{l,p,t} \in \mathbb{B} \qquad \forall l \in L, p \in P, t \in T$$

$$islt_{l,p,t} \in \mathbb{B} \qquad \forall l \in L, p \in P, t \in T$$

$$newb_{l,t} \in \mathbb{B} \qquad \forall l \in L, t \in T$$

Const: $CONT^{\$} \in \mathbb{R}_+$

$$BLD^{\$} \in \mathbb{R}$$

$$UPG^{\$}_{l,p} \in \mathbb{R}_+ \qquad \forall l \in L, p \in P$$

$$
\begin{aligned}
&UTIL^{\$}_{l} \in \mathbb{R}_{+} && \forall l \in L\\
&RELEASE^{\$}_{o} \in \mathbb{R}_{+} && \forall o \in O\\
&LFORL^{\$}_{o,l} \in \mathbb{R}_{+} && \forall o \in O, l \in L\\
&POSS_{o,l} \in \mathbb{B} && \forall o \in O, l \in L\\
&REL_{o,l} \in \mathbb{B} && \forall o \in O, l \in L\\
&LOCAL_{o,l} \in \mathbb{B} && \forall o \in O, l \in L\\
&LINKED_{l,s} \in \mathbb{B} && \forall l \in L, s \in S\\
&SIZE_{o,t} \in \mathbb{Z}_{+} && \forall o \in O, t \in T\\
&CAPM_{l,t} \in \mathbb{Z}_{+} && \forall l \in L, t \in T\\
&CAPS_{l,t} \in \mathbb{Z}_{+} && \forall l \in L, t \in T\\
&INITLINE_{o,l} \in \mathbb{B} && \forall o \in O, l \in L\\
&INITSITE_{o,s} \in \mathbb{B} && \forall o \in O, s \in S\\
&FIXSITES_{o,s,t} \in \mathbb{B} && \forall o \in O, s \in S, t \in T\\
&FT_{l,t} \in \mathbb{B} && \forall l \in L, t \in T\\
&CAPNM_{l,t} \in \mathbb{Z} && \forall l \in L, t \in T\\
&CAPNS_{l,t} \in \mathbb{Z} && \forall l \in L, t \in T\\
&ASGN_{o,p} \in \mathbb{B} && \forall o \in O, p \in P\\
&UPGRABLE_{l,p} \in \mathbb{B} && \forall l \in L, p \in P\\
&ILT_{l,p} \in \mathbb{B} && \forall l \in L, p \in P\\
&TUTIL_{l,t} \in \mathbb{R} && \forall l \in L, t \in T
\end{aligned}
$$

Next, we present our complete SMT formula. We want to mention that inherently, an SMT formula does not include an objective function and that we are thus abusing notation in the following. By including the objective function, we want to express that we can simulate the optimization process by including the sum of the objectives and demanding its value to be below a certain threshold. This is further discussed in section 5.3.3.

$$\min \sum_{t\in T}\sum_{l\in L} uaux_{l,t} \cdot UTIL^{\$}_{l} \cdot \omega_1 \tag{3.10}$$

$$+\sum_{o\in O}\sum_{l\in L} ITE(buyrel_{o,l}, RELEASE^{\$}_{o} \cdot \omega_2, 0) \tag{3.11}$$

$$+\sum_{t\in T}\sum_{o\in O}\sum_{l\in L} ITE(asgn_{o,l,t} \wedge \neg LOCAL_{o,l} \cdot \omega_3, LFORL^{\$}_{o,l}, 0) \tag{3.12}$$

$$+\sum_{t\in T_{>0}}\sum_{o\in O}\sum_{l\in L} ITE(aaux_{o,l,t}, CONT^{\$} \cdot \omega_4, 0) \tag{3.13}$$

$$+\sum_{t\in T}\sum_{p\in P}\sum_{l\in L} ITE(upgr_{l,p,t}, UPG \cdot \omega_5 \cdot 2^{|T|-t} + newb_{l,t} \cdot BLD^{\$}, 0) \tag{3.14}$$

$$\text{s.t.} \sum_{l\in L} prod_{o,l,t} = SIZE_{o,t} \qquad \forall o \in O, t \in T \tag{3.15}$$

$$\sum_{o\in O} prod_{o,l,t} \leq \sum_{i=0}^{t} ITE(newb_{l,t-i}, CAPNM_{l,i}, 0) + CAPM_{l,t} \qquad \forall l \in L, t \in T \tag{3.16}$$

$$INITLINE_{o,l} \rightarrow asgn_{o,l,0} \qquad \forall o \in O, l \in L \tag{3.17}$$

$$asgn_{o,l,t} \rightarrow actv_{l,t} \qquad \forall o \in O, l \in L, t \in T \tag{3.18}$$

$$\neg asgn_{o,l,t} \rightarrow (prod_{o,l,t} = 0) \qquad \forall o \in O, l \in L, t \in T \tag{3.19}$$

$$actv_{l,t} \rightarrow \bigvee_{o\in O} asgn_{o,l,t} \qquad \forall l \in L, t \in T \tag{3.20}$$

$$asgn_{o,l,t} \rightarrow \bigvee_{p\in P} (ASGN_{o,p} \wedge islt_{l,p,t}) \qquad \forall o \in O, l \in L, t \in T \tag{3.21}$$

$$(\neg REL_{o,l} \wedge asgn_{o,l,t}) \rightarrow buyrel_{o,l} \qquad \forall o \in O, l \in L, t \in T \tag{3.22}$$

$$(asgn_{o,l,t} \wedge LINKED_{l,s}) \rightarrow atsite_{o,s,t} \qquad \forall o \in O, l \in L, s \in S, t \in T \tag{3.23}$$

$$atsite_{o,s,t} \rightarrow \bigvee_{l \in L} (asgn_{o,l,t} \wedge LINKED_{l,s}) \qquad \forall o \in O, s \in S, t \in T \tag{3.24}$$

$$INITSITE_{o,s} \rightarrow atsite_{o,s,0} \qquad \forall o \in O, s \in S \tag{3.25}$$

$$FIXSITES_{o,s,t} \rightarrow atsite_{o,s,t} \qquad \forall o \in O, s \in S, t \in T \tag{3.26}$$

$$\begin{aligned} & actv_{l,t} \rightarrow (uaux_{l,t} \geq TUTIL_{l,t} - \frac{\sum\limits_{o \in O} prod_{o,l,t}}{CAPM_{l,t} + CAPNM_{l,t} + \varepsilon}) \\ & \wedge (uaux_{l,t} \geq \frac{\sum\limits_{o \in O} prod_{o,l,t}}{CAPM_{l,t} + CAPNM_{l,t} + \varepsilon} - TUTIL_{l,t}) \qquad \forall l \in L, t \in T \end{aligned} \tag{3.27}$$

$$\neg actv_{l,t} \rightarrow (uaux_{l,t} = 0) \qquad \forall l \in L, t \in T \tag{3.28}$$

$$aaux_{o,l,t} \leftrightarrow (asgn_{o,l,t-1} \oplus asgn_{o,l,t}) \qquad \forall o \in O, l \in L, t \in T_{>0} \tag{3.29, 3.30}$$

$$upgr_{l,p,t} \rightarrow UPGRABLE_{l,p} \qquad \forall l \in L, p \in P, t \in T \tag{3.31}$$

$$islt_{l,p,t} \rightarrow \neg lwu_{l,t} \qquad \forall l \in L, p \in P, t \in T \wedge ILT_{l,p} \tag{3.32}$$

$$islt_{l,p,t} \rightarrow islt_{l,p,t-1} \vee upgr_{l,p,t} \qquad \forall l \in L, p \in P, t \in T \wedge \neg ILT_{l,p} \tag{3.33}$$

$$islt_{l,p,0} \rightarrow upgr_{l,p,0} \qquad \forall l \in L, p \in P \wedge \neg ILT_{l,p} \tag{3.34}$$

$$actv_{l,t} \rightarrow actv_{l,t-1} \vee newb_{l,t} \vee FT_{l,t} \qquad \forall l \in L, t \in T_{>0} \tag{3.35}$$

$$\underset{p \in P}{ExactlyOne}(islt_{l,p,t}) \qquad \forall l \in L, t \in T \tag{3.36}$$

$$upgr_{l,p,t} \rightarrow islt_{l,p,t} \qquad \forall l \in L, p \in P, t \in T \tag{3.37}$$

$$lwu_{l,t} \leftrightarrow \bigvee_{p \in P} \bigvee_{i=0}^{t} upgr_{l,p,i} \qquad \forall l \in L, t \in T \tag{3.38}$$

$$newb_{l,t} \leftrightarrow \bigvee_{p \in P} upgr_{l,p,t} \wedge \neg lwu_{l,t-1} \wedge ILT_{l,P_0} \qquad \forall l \in L, t \in T_{>0} \tag{3.41}$$

$$newb_{l,0} \leftrightarrow \bigvee_{p \in P} upgr_{l,p,0} \wedge ILT_{l,P_0} \qquad \forall l \in L \tag{3.42}$$

$$\bigvee_{p \in P} \bigvee_{i=max\{t-BT,0\}}^{t-1} upgr_{l,p,i} \rightarrow \neg newb_{l,t} \qquad \forall l \in L, t \in T \tag{3.43}$$

$$aaux_{o,l,t} \in \mathbb{B}_+ \qquad \forall o \in O, l \in L, t \in T_{>0}$$

$$uaux_{l,t} \in \mathbb{R}_+ \qquad \forall l \in L, t \in T$$

$$prod_{o,l,t} \in \mathbb{R}_+ \qquad \forall o \in O, l \in L, t \in T$$

$$asgn_{o,l,t} \in \mathbb{B} \qquad \forall o \in O, l \in L, t \in T$$

$$atsite_{o,s,t} \in \mathbb{B} \qquad \forall o \in O, s \in S, t \in T$$

$$actv_{l,t} \in \mathbb{B} \qquad \forall l \in L, t \in T$$

$$buyrel_{o,l} \in \mathbb{B} \qquad \forall o \in O, l \in L$$

$$upgr_{l,p,t} \in \mathbb{B} \qquad \forall l \in L, p \in P, t \in T$$

$$islt_{l,p,t} \in \mathbb{B} \qquad \forall l \in L, p \in P, t \in T$$

$$newb_{l,t} \in \mathbb{B} \qquad \forall l \in L, t \in T$$

Const: $CONT^{\$} \in \mathbb{R}_+$

$$BLD^{\$} \in \mathbb{R}$$

$$UPG^{\$}_{l,p} \in \mathbb{R}_+ \qquad \forall l \in L, p \in P$$

$$UTIL^{\$}_{l} \in \mathbb{R}_+ \qquad \forall l \in L$$

$$RELEASE^{\$}_{o} \in \mathbb{R}_+ \qquad \forall o \in O$$

$$LFORL^{\$}_{o,l} \in \mathbb{R}_+ \qquad \forall o \in O, l \in L$$

$POSS_{o,l} \in \mathbb{B}$	$\forall o \in O, l \in L$
$REL_{o,l} \in \mathbb{B}$	$\forall o \in O, l \in L$
$LOCAL_{o,l} \in \mathbb{B}$	$\forall o \in O, l \in L$
$LINKED_{l,s} \in \mathbb{B}$	$\forall l \in L, s \in S$
$SIZE_{o,t} \in \mathbb{Z}_+$	$\forall o \in O, t \in T$
$CAPM_{l,t} \in \mathbb{Z}_+$	$\forall l \in L, t \in T$
$CAPS_{l,t} \in \mathbb{Z}_+$	$\forall l \in L, t \in T$
$INITLINE_{o,l} \in \mathbb{B}$	$\forall o \in O, l \in L$
$INITSITE_{o,s} \in \mathbb{B}$	$\forall o \in O, s \in S$
$FIXSITES_{o,s,t} \in \mathbb{B}$	$\forall o \in O, s \in S, t \in T$
$FT_{l,t} \in \mathbb{B}$	$\forall l \in L, t \in T$
$CAPNM_{l,t} \in \mathbb{Z}$	$\forall l \in L, t \in T$
$CAPNS_{l,t} \in \mathbb{Z}$	$\forall l \in L, t \in T$
$ASGN_{o,p} \in \mathbb{B}$	$\forall o \in O, p \in P$
$UPGRABLE_{l,p} \in \mathbb{B}$	$\forall l \in L, p \in P$
$ILT_{l,p} \in \mathbb{B}$	$\forall l \in L, p \in P$
$TUTIL_{l,t} \in \mathbb{R}$	$\forall l \in L, t \in T$

We obtain the reduced models by replacing constraint 3.21 with constraint 3.2, constraint 3.16 by 3.1, adjusting constrain 3.36 and by leaving out constraints 3.29 to 3.43. Addtionally, parts 3.13 and 3.14 of the objective function are deactivated.

3.4.1 Descriptions of Variables and Constants

Below, we give a reference for all variables and constants that are used throughout all formulations.
Variables:

- $uaux_{l,t}$: Auxiliary variable, reflects deviation from target utilization of a line l at time period t.
- $prod_{o,l,t}$: Amount of parts from an order o that are produced on a line l at time period t.
- $asgn_{o,l,t}$: Decides whether any part of an order o is produced on line l at time period t.
- $atsite_{o,s,t}$: Decides whether an order o is produced at production site s at time period t.
- $actv_{l,t}$: Decides whether line l is active (i.e. used) at time period t.
- $buyrel_{o,l}$: Decides whether a customer release has to be acquired for assigning an order o to a line l.
- $aaux_{o,l,t}$: Auxiliary variable, reflects whether an order o was continuously assigned to a line l in between two time periods $t-1$ and t.
- $upgr_{l,p,t}$: Decides whether a line l is to be upgraded to configuration p at time period t.
- $islt_{l,p,t}$: Decides whether a line l has a certain configuration p during time period t.
- $newb_{l,t}$: Decides whether a line l is upgraded from P_0 to some other configuration p in time period t.

Constants:

- $UTIL^{\$}_{l}$: Costs for not reaching the target production percentage on any line l which is given by the ratio between $CAPS_{l,t}$ and $CAPM_{l,t}$ as well as between $CAPNS_{l,t}$ and $CAPNM_{l,t}$.
- $RELEASE^{\$}_{o}$: Costs for not having a customer release when assigning an order o to any line.
- $LFORL^{\$}_{o,l}$: Costs for not honoring local for local regarding any assignment of an order o to a line l.
- $CONT^{\$}$: Costs for changing the assignment of orders between lines.
- $UPG^{\$}_{l,p}$: Costs for upgrading a line l to a certain type p.
- $BLD^{\$}$: Costs for constructing a new line.
- $POSS_{o,l}$: Indicates whether an order o can be technically assigned to a line l in the reduced problem.
- $REL_{o,l}$: Indicates whether there is a customer release already present for the assignment of an order o to a line l.
- $LOCAL_{o,l}$: States whether the delivery triad of an order o is local to that of a line l.
- $LINKED_{l,s}$: States whether a line l is stationed at site s.
- $SIZE_{o,t}$: Number of parts that are to be produced for order o in time period t.
- $CAPM_{l,t}$: Number of parts that a line l is able to maximally produce in time period t.
- $CAPS_{l,t}$: Target production value of a line l in timestep t, always below or equal $CAPM_{l,t}$.
- $CAPNM_{l,t}$: Number of parts that a line l is able to maximally produce in time period t after being constructed.
- $CAPNS_{l,t}$: Number of parts that a line l is targeted to produce in time period t after being constructed.
- $INITLINE_{o,l}$: Orders may initially be already be produced on a line during planning or this assignment of an order to a line is given otherwise. This binary constant reflects whether an order o is initially fixed to a line l.
- $INITSITE_{o,s}$: Orders may initially be already produced at a site. This binary constant reflects whether an order o is initially fixed to a site s.
- $FIXSITES_{o,s,t}$: Indicates whether an order o is fixated for production to a site s during a time period t.
- $ASGN_{o,p}$: Indicates whether an order o can be produced on an arbitrary line of type p.
- $ILT_{l,p}$: Indicates the initial configuration p of a line l in time period 0.
- $UPGRABLE_{l,p}$: Indicates whether a line l can be upgraded to a configuration p.
- P_0: The “empty” linetype, used for lines that are not constructed yet.

- $FT_{l,t}$: For a line l that can start production only later after the initial time period, this constant becomes true at the first time period at which the production amount of a line, i.e. $CAPNM_{l,t}$ becomes greater than 0.
- $TUTIL_{l,t}$: Percentage of utilization that is to be reached for a line l in a time period t.

3.5 Challenge: The Size of P

The specific production lines in the given optimization problem are special in that they are designed to be very modular regarding their configuration. A line configuration is defined by the presence or absence of line features. One can interpret a line configuration as a boolean vector $fvec$ in the length of all possible features n, with an index $fvec[i], i \in \{0, \ldots, n-1\}$ evaluated to $True$ if the feature is present and to $False$ if it is not. Particularly, there are no interdependencies between line features, i.e. adding a feature to a line will not make it impossible to additionally add another feature that was also previously possible to be added.
In theory this means that every vector of features has to be regarded when determining the set of all line types P that may be relevant for the model, resulting in an exponential blowup in the size of n. In our given problem, the lines have a total of 35 features relevant for determining whether an order can be produced on a line or not. In practice however, most lines have a small set of features that can be added, with the highest number of features that can be added per line being only six. As the possible variation in line configurations is this small, we decided that simply enumerating every possible line configuration for every line was not going to give as much blowup as theoretically expected. We found the number of all possible and unique line configurations over all lines to be around 300, which is a smaller by a great margin than the expected 2^{35} configurations and deemed acceptable for our problem.

3.6 Quality Evaluation of our Formulation

We want to briefly evaluate which aspects of our mathematical formulations may be further enhanced in the future.
Right now, by our design, the objective function is dominated by the cost of upgrades, as we found this to be an easy way to handle both the delay of upgrades time-wise and from the aspect of only upgrading what needs to be upgraded. We suggest revisiting the involved constraints in order to find an alternative way of modelling upgrades that is both as memory-efficient as well as precise as our variant, but is able to differently express the objective function so as to get rid of the exponential backoff.
From the standpoint of implementation, it may be desirable to directly propagate the values of constants in the impementation, so as to reduce the number of constraints and the number of variables that need to be decided. While some MIP solver like *SCIP* do this propagation to a certain degree automatically as part of their "presolving" routine, we cannot assume all solvers to implement this behaviour.
Regarding the coupling of assignment variables and production-amount variables in constraint 3.19, it may be desirable to find a "tighter" formulation that does not need the multiplicative factor of $CAPM_{l,t}$ and $CAPNM_{l,t}$. We assume that a reformulation of this constraint may help the perfomance of our solvers.

Chapter 4

Implementation

Besides finding a mathematical formulation of the problem stated by the Robert Bosch GmbH, one of the main goals of this work was to write a software tool that is capable of extracting the raw data given by the Robert Bosch GmbH in the form of a multi-sheet workbook, followed by constructing an object for both the MIP and the SMT formulations, invoking a solver on this object containing our parsed input data and writing out a solution in a processable form, if a solution exists.
The software implementation was done using the programming language Python in version 3.6. All of the source code was thoroughly commented for the generation of a documentation using the tool Doxygen [8]. A doxyfile is included in the Code folder.
Structurally, there are several components interacting with one another such as the parser, a benchmark generator and an abstract problem handler. We give an abstract overview of the control flow in a usual run of the software in figure 4.1: The main module is invoked with a set of options - detailed in section 4.3 - which include the path of the file needed for processing the problem. If no input file is specified via the options and the user has a graphical frontend, a file selection dialog is opened to select the needed file.
A data object, which we call a *DataBundle*, is created and invokes a call to the parser on the given file. The parser then extracts the necessary data from the given input files and passes them back to the DataBundle object, which in turn checks the extracted data for general consistency, such as non-empty data fields and correct typing, but also for inconsistent data, which we detail later.
After being returned the DataBundle object, the main module then invokes a call to a problem handler - SMT or MIP - in order to generate a mathematical problem of the according format. The created problems may optionally be exported to .smt2 or .mps files. Finally, the main method invokes a call to solve the generated problem, using a tool specified by the user via the options passed in the beginning. If a solution is found, it is then written out for the user to inspect.
For testing and benchmarking, artificial and randomized data - with regards to a seed - can be created using the provided benchmark generator. This will create a DataBundle object as well, and thus the program flow that we described is identical after the DataBundle object is returned. We will go into more detail on how this data is generated in the Performance Evaluation chapter. The structure of a benchmark run is depicted in figure 4.2.
We will now discuss the components of the software in more detail.

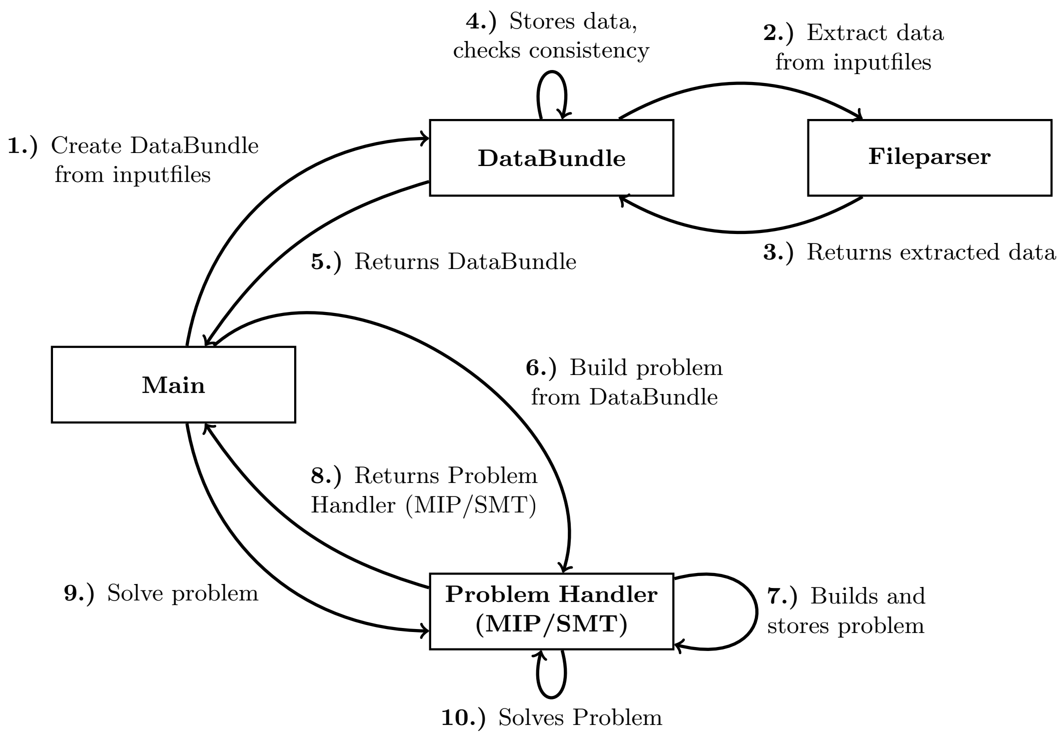
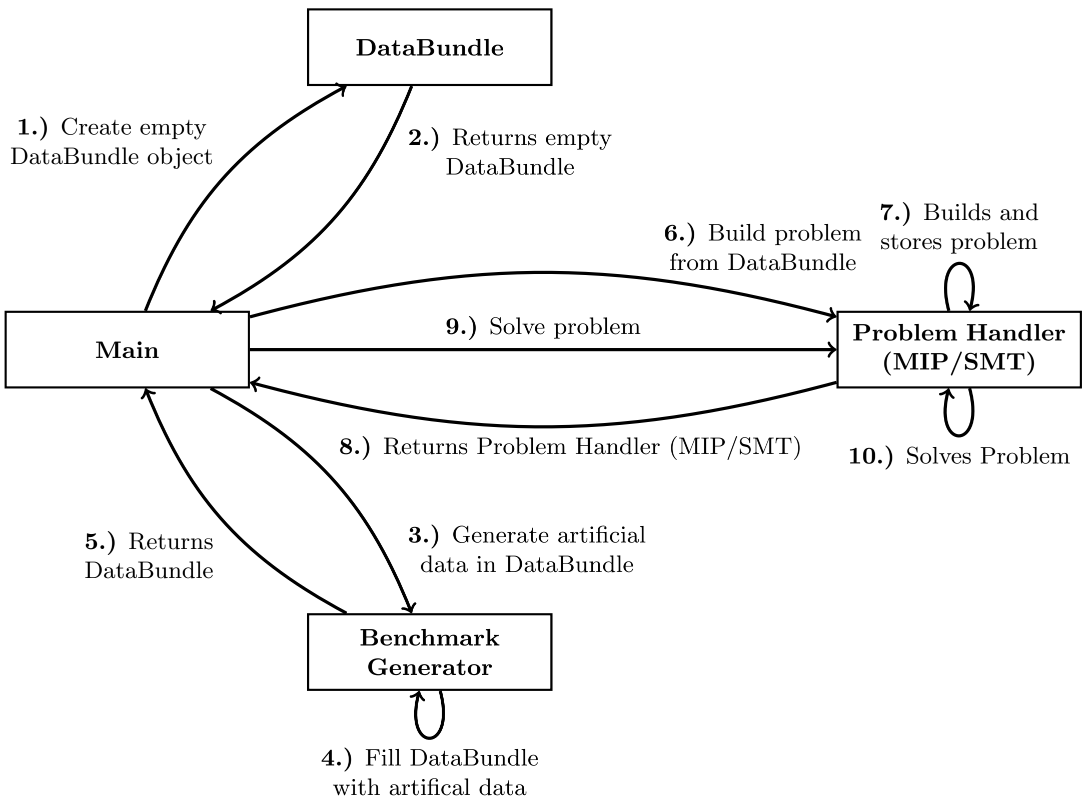

4.) Stores data, checks consistency
2.) Extract data from inputfiles
1.) Create DataBundle from inputfiles
DataBundle
Fileparser
5.) Returns DataBundle
3.) Returns extracted data
Main
6.) Build problem from DataBundle
8.) Returns Problem Handler (MIP/SMT)
9.) Solve problem
Problem Handler (MIP/SMT)
7.) Builds and stores problem
10.) Solves Problem

Figure 4.1: The simplified control flow throughout a normal run. After the last step, the main module invokes an output of the solution.



DataBundle
1.) Create empty DataBundle object
2.) Returns empty DataBundle
7.) Builds and stores problem
6.) Build problem from DataBundle
Main
9.) Solve problem
Problem Handler (MIP/SMT)
8.) Returns Problem Handler (MIP/SMT)
10.) Solves Problem
5.) Returns DataBundle
3.) Generate artificial data in DataBundle
Benchmark Generator
4.) Fill DataBundle with artifical data

Figure 4.2: The simplified control flow throughout a benchmark run. After the last step, the main module invokes an output of the solution.

4.1 Building the DataBundle Object

The data that is provided to us in order to generate our concrete problem is given in the form of a multi-sheet workbook. This workbook contains a total of ten sheets that are relevant for our problem, of which six are raw input data and four are exclusively used to detail the artificial cost parameters for our objective function. Additional to configuring rather natural parameters of the problem such as line capacities or possible upgrades for production lines, the user is able to control the following parameters during planning inside the input file:

- Assigning orders to lines and sites at the initial time period.
- Assigning orders to sites during any time period.
- Setting cost parameters for underutilization of each line.
- Setting cost parameters for buying customer releases for each order.
- Setting cost parameters for violating the concept of “local for local”.
- Setting the cost parameter for incontinuity of line assignments across time periods.
- Setting the cost parameter for constructing production lines.
- Setting the initial time period for planning.

In order to extract the necessary data, we implemented a parser module that first converted the file of .xlsb format into several files of .csv format - one for each relevant workbook sheet, using the library pyxlsb[9]. The mentioned parser is invoked by the DataBundle module, which checks the extracted data for consistency in regards to non-empty fields, correct typing and consistent naming of orders, lines and sites across the different input data sheets. Additionally, after all data is acquired, further checks are made which allow for the early detection of data inconsistencies which would result in unsatisfiability:
In the context of the reduced model, it is checked whether orders are initially assigned to lines or sites which are incapable of producing them. As no upgrades are possible in the reduced model, this would mean that the order can simply not be produced. Similarly, in the complete model, orders may be initially assigned to a line which even when upgraded is unable to produce the order or to sites of which no line is able to produce the order for the same reason. There may be orders that require a certain line configuration that none of the lines is able to provide, even when upgraded and lastly, all the lines that are capable to produce a certain type of order may have their capacities set to 0, i.e. shut down.
If any of the checks for the extended model are unsuccessful in that they find an inconsistency, the program is terminated at the end of the checks. The user is then able to extract the exact contradictions from the generated logs.

4.2 Creating and Solving the Mathematical Model

Given our required input data in form of a DataBundle object, the next step is to create an object that is able to generate, solve and print our mathematical problem. We have chosen to create an abstract *problem handler* class as a parent class for both the MIP and SMT formulae. This way, we can rely on a unified interface independent of the underlying problem formulation format. Apart from requiring the implementation of a few attributes, e.g. for timing, this class enforces the following methods: Creating the concrete model from a DataBundle object using *_create_problem_from_data_bundle()*, writing the created problem with *write_problem()*, solving the problem using a *solve()*-method and finally the ability to print a solution - if one exists - using *write_solution()*.

Our requirement for the implementation of both problem formulations was the ability to create problem objects of a common format for most solvers and the ability to directly interface as many common solvers as possible. These criteria were met by two libraries, which we will introduce in the following.

4.2.1 Implementation of SMT: pySMT

The established standard for describing SMT formulae is the SMT-lib standard[10] with the .smt2 file format. For the implementation of our SMT formula, we decided to use the python library *pySMT*[11], released under the Apache 2.0 license. The library provides interfaces to common SMT solvers such as *Z3*, *MATHSAT* and *CVC4* and an SMT-lib interface in order to interface any SMT-lib compatible solver. Additionally, created formulae can be output in the SMT-libs own *.smt2*-format.

4.2.2 Implementation of MIP: Pyomo

For the implementation of our MIP, we decided to use the python library *Pyomo*[12][13], released under the BSD license. The library provides interfaces to common LP- and MIP-solvers, such as *Gurobi*, *GLPK*, *SCIP* and *CPLEX*. Additionally, the output of a Pyomo object to common formats such as the AMPL-format *.nl* and the CPLEX *.lp*-format is supported.

4.3 User Parameters

In order to control parameters of the mathematical and the general behaviour of the software, the user is provided with two ways of configuration: a configuration file and command line options. The configuration file has a very simple structure, a sample file named *config.ini* is included in the *Code*-directory. Using this file, the user is able to control two things: General parameters for solver behaviour and parameters for creating a benchmark run with artificial data. The parameters are the following:

- **solution_gap**: If a solution was found that deviates at most this percentage from the optimal solution, a run is to be preemptively terminated and the current solution is to be output.
- **line_utilization_weight**: Weighing factor ω_1. Used to control the weight of line utilization in the objective function.
- **customer_release_weight**: Weighing factor ω_2. Used to control the weight of customer releases in the objective function.
- **local_for_local_weight**: Weighing factor ω_3. Used to control the weight of “local for local” in the objective function.
- **line_continuity_weight**: Weighing factor ω_4. Used to control the weight of line continuity in the objective function.
- **upgrade_costs_weight**: Weighing factor ω_5. Used to control the weight of upgrade costs in the objective function.

- **seed**: An integer number used for generating the seed for randomization. If a value of 0 is entered, the current timestamp is used to seed a benchmark run.
- **nOrders**: Number of orders for a benchmark run.
- **nLines**: Number of lines for a benchmark run.
- **nSites**: Number of production sites for a benchmark run.
- **nPeriods**: Number of time periods for a benchmark run.
- **nLineTypes**: Number of different line configurations for a benchmark run.

The command line options are used to control more general behaviour of the software. The following options are available:

- **{-h,--help}**: Show a help message displaying all options.
- **--tpz_file**: Path to the main data file in .xlsb format.
- **{-l, --disable_logging}**: Disable all logging.
- **{-c, --do_not_log_to_console}**: Disable forking the logging output to stderr.
- **--write_problem_files**: Write out .mps/.smt2 files after the according problems are generated.
- **--mip_solver**: Specify the name of the MIP solver to be used. All solvers mentioned in the section "Benchmark Criteria" are possible.
- **--smt_solver**:Specify the name of the SMT solver to be used. All solvers mentioned in the section "Benchmark Criteria" are possible.
- **--solution_filename**: Specify the name of the solution folder which is also the prefix to all of its file names, if a solution exists. If no name is specified, the current timestamp is used.
- **{-v, --verbose}**: Include all debug messages in the log.
- **--only_write_problem_file**: Converts the input files to .smt/.mps files without invoking the solving process.
- **--only_check_satisfiability**: Stop the solving process once a feasible solution was found.
- **--config_file**: Path to a configuration file to be used for the run.
- **--run_benchmark**: Flag to indicate whether a benchmark is supposed to be run. User needs to provide a config file specifying the parameters for the benchmark. See --config_file.
- **--reduced_problem**: Flag, only creates the reduced problem if set.
- **--smt_gap**: Triggers optimization using the selected smt solver. Maximal derivation from an optimal solution value that is to be searched for.

4.4 Output of a Run

For every run, output files are produced. This output is placed inside a timestamped folder as a subdirectory of the *outputfiles* directory in the *Code* directory, if not specified differently by the input options. Every run produces a log - if not explicitly disabled. Problem files in .smt2-format for the SMT problem formulation or in .mps-format for the MIP problem formulation can be written, either alongside a run in which they are also solved or as the only purpose of the run. If a solver was invoked and a solution was found, multiple files of *.csv* format are written out in order to visualize the results of the run:

- A file indicating which lines are active in which time periods.
- A file that details to which concrete configuration lines have to be upgraded to in which time period.
- A file specifying for which combination of orders and lines customer releases have to be acquired.
- For each time period a file that details how many parts of which order are to be produced on which line.

An additional requested feature by the Robert Bosch GmbH was a file of .csv format that details which orders can technically be produced on which set of lines, using their initial line configuration and the features needed for producing the order as a basis. This file is generated during every run.

Chapter 5

Performance Evaluation

One of the goals of this work was to test and compare the performance of multiple SMT- and MIP-solvers on our given problem. To this end, we only generated artificial test sets due to hardware restrictions on the machines provided by the Robert Bosch GmbH and the restriction that the data may not leave the infrastructure of the Robert Bosch GmbH. For randomization, we use Python's own *random* module and give the user the ability to control the seed in a configuration file. We will provide the seeds that we are using for each benchmark run along with the results, so that the reader has the ability to verifiy the results themselves. Please note that when these benchmarks were performed, the model did not include constraint 3.20, which means the reader has to comment out the regarding constraint in the problem handlers in order to obtain the same results.
We will first discuss how we generate our test data, then motivate which variations of the problem formulation we are going to test and finally provide benchmark results for each of the given test sets.

5.1 Generating Randomized Test Sets

As we have mentioned in our Implementation chapter, our *benchmark_generator* module can be used to generate artificial DataBundle objects in order to test and benchmark our problem formulations. We have already learned that our problem consists of five different dimensions: Orders, production lines, line configurations, production sites, and time periods. The user is able to specify the size of each dimension in a configuration file that is passed when invoking the main module. In this section, we will detail how the data is generated in order to give the reader the ability to better evaluate the benchmark results. For this, we need to go over the values of each constant.

5.1.1 Generating Line Configurations

We first need to discuss how we handle line types internally, as some constants can only be created in a consistent way if the handling of line configurations is clear. If we e.g. choose a random value for a constant $ASGN_{o,p}$, we state that an order o can be produced on an arbitrary line of type p. However, lines are upgradeable and are able to produce strictly more different kinds of products afterwards. This means that if a configuration p_1 is suitable to produce order o, then also the configuration p_2 should be suitable to produce o if configuration p_2 is the result of an upgrade from p_1. Since it would be rather tedious to keep track of such dependencies

by simply generating arbitrary and non-specific line configurations, we decided to use the same approach as we use for encoding line types when handling real data.
As we have already learned, a line configuration is the collection of present and absent features. Internally, we encode this by a boolean vector, with an entry for each feature. We then simply generate random vectors of same length, which are then our line configurations. As we do want it to be possible for a lot of line types to be upgradeable into one another, we restrict the length of these vectors to $\lceil log_2(nLineTypes) \rceil$. The "empty" configuration, which is a vector of $False$ entries is always part of our line type set, as well as the full feature vector, of which all entries are $True$. The latter is added in order to ensure satisfiability of our benchmark data, as this adds the solution in which every line is upgraded to the full feature vector. Obviously, this means that the number of line configurations that the user can choose may not fall below 2. All other line configurations are created by generating random vectors and adding them to our line type set if they are not already part of it, until we have the desired number of vectors. As we set the length of our vectors to $\lceil log_2(nLineTypes) \rceil$, we can always ensure that this procedure terminates in expectation.
Before we move on, we observe that this vector encoding allows us to define a half order on this particular set of P as follows:

Definition 5.1.1 (The Half-Order $<$ on P)**.** Given two boolean vectors $p_1, p_2 \in P$. Let n be the length of these vectors. Let $v[i]$ be the i-th entry of vector v. We define the half-order $<$ as follows:

$$p_1 < p_2 \quad \Leftrightarrow \quad p_1[i] \leq p_2[i] \quad \forall i \in \{1, \ldots, n\} \wedge \exists i \in \{1, \ldots, n\} \text{ s.t. } p_1[i] < p_2[i]$$

5.1.2 Randomized Values for Constants

LINKED$_{\mathbf{l,s}}$: Every line is stationed at exactly one site, thus we randomly choose the site of each line.
LOCAL$_{\mathbf{o,l}}$: The information whether an order is local to the line that it is produced on is both dependent on the site that the line is located at as well as the delivery location of the order. As this delivery location is in theory completely arbitrary, we randomly choose a site from the set of all sites S which is then the site of which each line is local to the order. Note that each order is local to at least one production site by definition.
POSS$_{\mathbf{o,l}}$: In our reduced model, we utilize the constant $POSS_{o,l}$ which states whether an order can be technically produced on a line or not. For each pair of orders and lines we flip a coin whether this line is able to produce it or not. As our reduced problem can only be satisfied if each order can be produced on at least one line and in theory, each coin flip can result in an order not being assignable to a line, we additionally randomly determine a line for each order on which it can be produced.
REL$_{\mathbf{o,l}}$: Customer releases for producing orders on lines can be seen as being completely arbitrary. For this reason, we throw a coin for every pair of orders and lines whether there is a release present in order to produce the given order on the given line.
SIZE$_{\mathbf{o,t}}$: For the size of each order, we try to mimic the real data closely. It is often not the case that the time periods in which an order is to be produced span from time period 0 to time period $|T|-1$. Usually, there is a window of time periods in which the product is to be produced. Thus, for each order, we pick a random subinterval of $[0, |T|-1]$ which is supposed to be the production interval and randomly pick the size of the order out of the interval $[1, 100000]$ for each time period inside the interval.
ILT$_{\mathbf{l,p}}$: The intial linetype is randomly picked from the list of all linetypes for every line.
CAPM$_{\mathbf{l,t}}$: The maximal capacities of the lines are depending on the sizes of our orders: For

each order, we determine all lines on which the order can be produced - either initially or after upgrades - and determine a subset of these lines. Then, for each line and time period, we evenly distribute the size of the order on all of these lines for the current time period.
$\mathbf{CAPS_{l,t}}$: As the target utilization of a line in percent is given by dividing the target capacities of a line by the maximal capacities of this line. We generate our standard capacities by randomly picking a value from the set {0.5,0.6,0.7,0.8,0.9,1.0} and multiplying this value with the maximal capacities that we have generated before.
$\mathbf{CAPNM_{l,t}}$: We can check whether we have generated lines that are not constructed by inspecting the values of $ILT_{l,p}$. If there is at least one member in the set of “empty” lines, we choose one line at random from this subset. Additionally, we choose a line for which we have already calculated our capacities. For every time period, half of the capacities that this constructed line was assigned is now shifted towards the capacities of the line that is not yet constructed. As the capacities are calculated from the size of all orders and as there are exactly as many capacities in lines as there are parts to be produced, this ensures that this randomly selected line has to be constructed.
$\mathbf{CAPNS_{l,t}}$: Just like the standard capacities for already built lines, we randomly choose a percentage from the set {0.5,0.6,0.7,0.8,0.9,1.0}.
$\mathbf{UPGRABLE_{l,p}}$: Given our initial line type, we allow it to be upgraded to every other line type of our set P if the chosen candidate vector can be ordered above our initial line type with respect to our half-order $<$ defined in definition 5.1.1. This especially means that in our setting, each line can be upgraded to our full feature vector.
$\mathbf{ASGN_{o,p}}$: Based on the values of $ILT_{l,p}$ and $UPGRABLE_{l,p}$ we can now randomly determine to which configuration an order is supposed to be assignable. We simply collect all line types from $ILT_{l,p}$ and $UPGRABLE_{l,p}$, and for each order pick a line type from this set. This order is then assignable to this line type and to all of its possible upgrades.
$\mathbf{INITLINE_{o,l}}$, $INITSITE_{o,s}$: Next, we look at generating assignments of orders to sites and lines at time period 0. In the core, the mechanism for determining initial assignments is the same in both the complete and the reduced problem: For each order, a random line is picked from a list of lines on which it can be produced and for this line, a coin is tossed to tell whether the order is initially assigned to it or not. If the line is not assigned, a random site from a list of compatible sites is picked and another coin is tossed to determine whether the order is afixed to the site. A site is compatible if at least one line of this site is able to produce the order. The difference between the complete and the reduced model is how the list of compatible lines and sites is determined: In the reduced version, we determine compatible lines and sites by using the values of our constant $POSS_{o,l}$ while in the complete model, we utilize the values of $ILT_{l,p}$ and $ASGN_{o,p}$ in order to determine whether an order could be produced on a line initially or after an upgrade.
$\mathbf{P_0}$: Our empty line type is the vector of length $\lceil log_2(nLineTypes)\rceil$ that only consists of $False$ entries.
$\mathbf{FIXSITES_{o,s,t}}$: We decide not to fixate any orders to certain sites, as in our original data, only 3 orders were fixated beyond the first time period.
BT: For the construction time of every line, we choose a single value randomly from the set {1,2,3}.
$\mathbf{FT_{l,t}}$: The values of this constant are completely dependent on the values of the (maximal) capacity values, which we randomly generated as described above. As with real data, the first time of production is the time period in which the production amount switches from 0 to some nonzero value for the first time.
$\mathbf{UTIL_l^{\$}}$: For each line, we choose a random integer value between 1 and 100.
$\mathbf{RELEASE_o^{\$}}$: For each order, we choose a random integer value between 1 and 100.

$\mathbf{LFORL}^{\$}_{\mathbf{o,l}}$: For each pair of orders and lines, we choose a random integer value between 1 and 100.
$\mathbf{CONT}^{\$}$: We choose a random integer value between 1 and 100.
$\mathbf{UPG}^{\$}_{\mathbf{l,p}}$: For each feature, we choose a value between 1 and 10 and accumulate the costs for each vector.
$\mathbf{BLD}^{\$}$: We choose a random integer value between 1 and 100.

5.1.3 Disproportional Memory Usage during Runtime

During development, we noticed that the libraries pySMT and pyomo consumed disproportional amounts of RAM both when writing out a problem file and when invoking a solver with the dimensions of the original problem data. For the SMT case, this resulted in us being unable to simulate a run with artificial data based on the dimensions of the original problem, as our most performant machine had only 32 Gigabytes of memory, which were exceeded.
When using Pyomo, interfacing a solver with a problem on the basis of the dimensions of our original data, we reached almost 24 Gigabyte of RAM usage. These dimensions were: 1093 orders, 41 production lines, 5 production sites, 13 time periods and around 300 line configurations. Since the hardware that we used for creating our benchmarks was only equipped with 16 Gigabytes of RAM - specified in more detail in the next section -, this resulted in us being unable to test the runtime of our problem on instances of a size similar to our original problem.

5.2 Benchmark Criteria

As we have both a SMT as well as a MIP formulation at hand, we would like to use this opportunity to compare both kinds of solvers with one another as well as compare the solvers of both categories with one another. As we have already mentioned in our preliminaries, SMT solvers are not inherently built to optimize problems but rather to find a satisfying solution of arbitrary quality to a given problem. This is why we only compare MIP and SMT solvers in the criteria of finding the first satisfying solution to a given problem.
When comparing the solvers of the two categories with one another, we are mainly interested in their performance on finding single satisfying solutions for problems of increasing dimensions, while in the case of MIP solvers we are also interested in the time needed between finding a solution of a given quality (gap) and finding the optimal solution to a problem.
In order to ensure that the performance of a solver is not enhanced or worsened disproportionally by the concrete data of an instance, we generate three differently seeded test sets for each benchmark test and average the results for each solver. Our time measurement starts by passing the input file to the solver and ends as soon as the program terminates. This especially means that we ignore the runtimes output by the solvers themselves.
As we have already mentioned, we will provide the seeds for each run such that the reader may verify the results themselves. Due to time limitations, we set the timeout for each solver run to 10 minutes (600 seconds). All benchmarks are made on an Intel i7-7500U with 16GB of RAM (no swap) running Arch Linux. As not all solvers allow for parallelization, we run every benchmark on a single thread. To obtain the runtime, we preceed every call to the solver with the command *time timeout 600*, i.e. we measure the time between invoking the given tool and the time it takes for a solver to return control.
Due to time constraints, we are unable to compare the performance of the *compressed* variant to that of our implmented *distributed* variant. We suggest this evaluation to be done in a future work. We compare the following solvers:
MIP:

Name	Seeds	nOrders	nLines	nSites	nPeriods	nLineTypes
Base	1,2,3	125	15	5	5	75
Ord.	1,2,3	250	15	5	5	75
Line	1,2,3	125	30	5	5	75
Site	1,2,3	125	15	10	5	75
Time	1,2,3	125	15	5	10	75
Type	1,2,3	125	15	5	5	150

Figure 5.1: Complete list of all benchmark sets for our *doubling dimensions* benchmark using artifical test data. Each set is defined by 3 differently seeded instances.

- Gurobi 8.0.1 [14]
- SCIP 6.0.0 [15][16]
- CPLEX 12.8.0.0 [17]
- GLPK (glpsol) 4.65 [18]

SMT:

- Z3 4.7.1 [19]
- MATHSAT 5.2.10 [20]
- CVC4 1.6 [21]

We run two different kinds of benchmarks, each with a different goal. The first benchmark is structured as follows. We choose a fixed set of dimensions as our *Base* set and iteratively double one of the given dimensions. The sets are listed in figure 5.1. We choose the size of the dimensions such that each of our tested solvers at least passes the benchmark on the *Base* set. On each set, we then measure the time it takes for a solver to find a single satisfying solution, a solution that is at most 50%/10%/2% away from the optimal solution and the time to find an optimal solution.
The purpose of this benchmark is to compare the performance of our tested solvers under different criteria. On the basis of the benchmark data, we can also make assumptions on which dimensions of our problem are the most critical regarding runtime.

For our second benchmark, we also choose three test sets of which the dimensions are fractions of the dimensions of the input data given by the Robert Bosch GmbH.

Finally, we investigate the option to optimize our problem using the SMT solver Z3.

5.3 Benchmark Results

In the following, we list and discuss the results of our benchmarks.

5.3.1 First Benchmark: Doubling Dimensions

The amount of time to find a single satisfying solution, depicted in the tables of figures 5.2 and 5.3 and visualized in figure 5.8, yields a noticeable result: The solvers Gurobi and CPLEX are both faster than any of the SMT solvers. The solvers reported these solutions to be found by

Testset	GUROBI	CPLEX	SCIP	GLPK	Z3	MSAT	CVC4
Base	0.31	0.89	18.88	386.29	6.66	51.26	77.55
Ord.	0.51	1.55	47.3	$600^{\dagger\dagger\dagger}_{+}$	10.79	35.1	383.29
Line	2.12	1.78	$600^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	35.1	160.23	341.71
Site	0.87	1.03	20.48	$600^{\dagger\dagger\dagger}_{+}$	8.32	51.41	83.61
Time	1.1	2.72	90.76	$600^{\dagger\dagger\dagger}_{+}$	21.18	150.65	399.83
Type	0.37	1.16	28.92	$348.53^{\dagger}$	9.78	64.79	115.19
Sum	5.3	9.16	806.35_{+}	3134.83_{+}	91.85	623.03	1401.19

Figure 5.2: Benchmark *doubling dimensions*: Finding a single satisfying solution. Values measured in seconds. Runtime averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Testset	GUROBI	CPLEX	SCIP	GLPK	Z3	MSAT	CVC4
Base	0.39	2.69	56.64	1158.89	19.99	153.78	232.67
Ord.	1.54	2.66	141.9	$1800^{\dagger\dagger\dagger}_{+}$	32.38	434.07	1149.87
Line	6.37	5.36	$1800^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	105.32	480.69	1025.14
Site	2.63	3.1	61.45	$1800^{\dagger\dagger\dagger}_{+}$	24.98	154.23	250.83
Time	3.3	8.18	272.29	$1800^{\dagger\dagger\dagger}_{+}$	63.55	451.96	1199.49
Type	1.13	3.5	86.78	$1297.07^{\dagger}$	29.34	194.38	230.38
Sum	15.9	27.49	2419.06_{+}	9655.96_{+}	275.56	1869.11	4088.38

Figure 5.3: Benchmark *doubling dimensions*: Finding a single satisfying solution. Values measured in seconds. Runtime summed over all instances of a set. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

heuristics.
The runtimes for finding a solution that is at most 50% away from the optimal solution is depicted in figure 5.4 and visualized in figure 5.9. Similarly, the runtimes for the gap of 10% and 2% are given in the tables of figures 5.5 and 5.6, and visualized in figures 5.10 and 5.11. The runtimes to find an optimal solution can be found in the table of figure 5.7 with a visualization of the runtime in figure 5.12 We additionally provide a runtime overview of all MIP-solvers over all of our types of benchmarks in figure 5.13. From this, one can clearly see that for almost all solvers, a solution of a quality that derivates by at most 10% from the optimal solution is found rather quickly, while trying to find a solution that derivates by at most 2% is in part time intensive and finding an optimal solution is not realistic for most instances and solvers, even for the given, rather small data set. To better visualize this observation, we have plotted the reported solution quality of the solver gurobi on the three instances of the basic benchmark set in figure 5.14. It is noticeable how long the solver needs to close the last percent of derivation between its currently found and the optimal solution.

5.3.2 Second Benchmark: Converging to Original Dimensions

For our second benchmark, we want to investigate the runtime behaviour on instances whose dimensions are fractions of the original input data. We created three test sets, again each consisting of three instances, which are to represent a quarter, half and three fourths of our original dimensions of around 1100 orders, 40 production lines, 5 production sites, 13 time

Testset	GUROBI		CPLEX		SCIP		GLPK	
Base	2.42	7.28	11.2	33.6	18.88	56.64	386.29	1158.89
Ord.	5.73	17.2	19.91	59.74	47.3	141.9	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Line	10.71	32.15	128.77	386.32	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Site	2.81	8.44	21.74	65.24	20.48	61.45	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Time	1.99	5.98	47.78	143.35	90.76	272.29	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Time	3.41	10.23	17.98	53.96	28.92	86.78	$348.53^{\dagger}$	$1297.07^{\dagger}_{+}$
Sum	27.09	81.28	247.4	742.21	806.35_{+}	2419.06_{+}	3134.83_{+}	9655.96_{+}

Figure 5.4: Benchmark *doubling dimensions*: Finding a solution whose objective value is at most 50% away from the optimal solution. Columns: Runtime averaged over passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Testset	GUROBI		CPLEX		SCIP		GLPK	
Base	3.09	9.27	11.20	33.60	18.88	56.64	386.29	1158.89
Ord.	6.33	18.99	19.91	59.74	47.30	141.90	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Line	12.91	38.75	128.77	386.32	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Site	3.49	10.47	21.74	65.24	20.48	61.45	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Time	14.02	42.06	173.35	520.05	90.86	272.29	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Type	4.12	12.36	17.987	53.96	28.927	86.78	$348.535^{\dagger}$	$1297.07^{\dagger}_{+}$
Sum	43.96	131.90	372.97	1118.91	806.35_{+}	2419.06_{+}	3134.83_{+}	9655.96_{+}

Figure 5.5: Benchmark *doubling dimensions*: Finding a solution whose objective value is at most 10% away from the optimal solution. Columns: Runtime averaged over passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Testset	GUROBI		CPLEX		SCIP		GLPK	
Base	16.17	48.52	11.2	33.6	154.75	464.24	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Ord.	6.31	18.95	19.91	59.74	136.02	408.07	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Line	$324.42^{\dagger}$	$1248.85^{\dagger}_{+}$	307.87	923.6	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Site	32.65	97.97	37.64	112.93	257.89	773.68	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Time	59.62	178.86	173.35	520.05	$71.33^{\dagger}$	$742.66^{\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Type	21.53	64.61	19.41	58.25	219.96	659.88	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Sum	460.72_{+}	1657.76_{+}	569.39	1708.17	1439.95_{+}	3048.53_{+}	$3600^{\dagger\dagger\dagger}_{+}$	10800_{+}

Figure 5.6: Benchmark *doubling dimensions*: Finding a solution whose objective value is at most 2% away from the optimal solution Columns: Runtime average over passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Testset	GUROBI		CPLEX		SCIP		GLPK	
Base	260.35	781.05	$188.11^{\dagger}$	$976.22^{\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Ord.	$20.5^{\dagger}$	$641^{\dagger}_{+}$	$60.5^{\dagger}$	$721^{\dagger}_{+}$	$213.61^{\dagger\dagger}$	$1413^{\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Line	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Site	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Time	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Type	138.117	414.35	$243.26^{\dagger\dagger}$	$1443.26^{\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Sum	2218.96_{+}	7236.40_{+}	2291.87_{+}	8540.48_{+}	3213.61_{+}	10413.61	3600_{+}	10800_{+}

Figure 5.7: Benchmark *doubling dimensions*: Finding an optimal solution. Values measured in seconds. Columns: Runtime averaged over passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

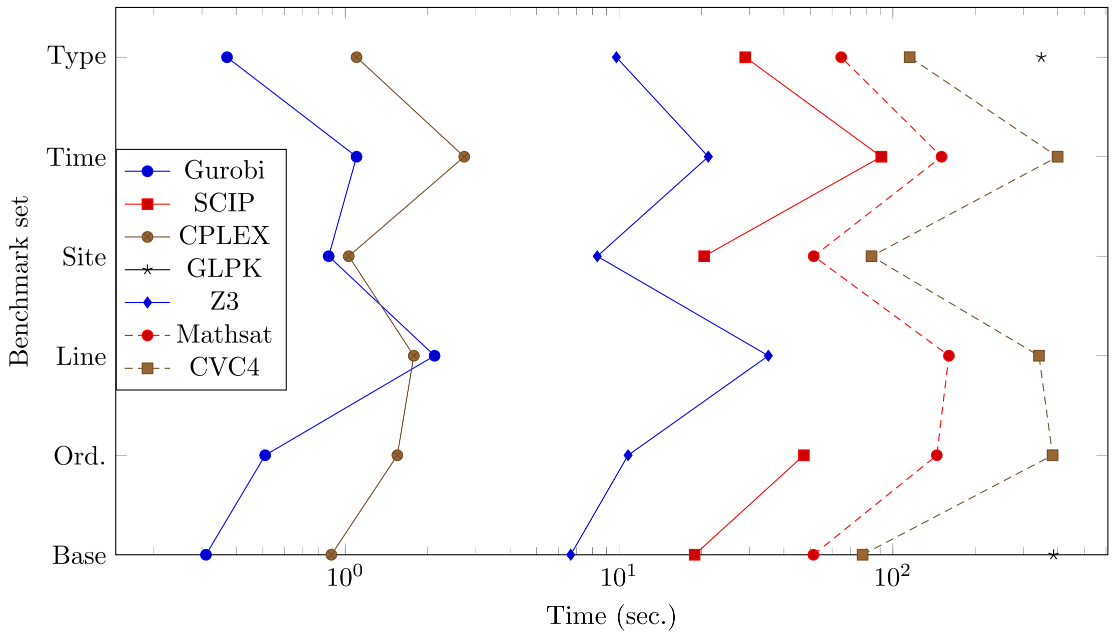


Figure 5.8: Average time needed to find a single satisfying solution on the benchmark for doubling dimensions. Missing points for a set indicate a timeout on all instances of a set.

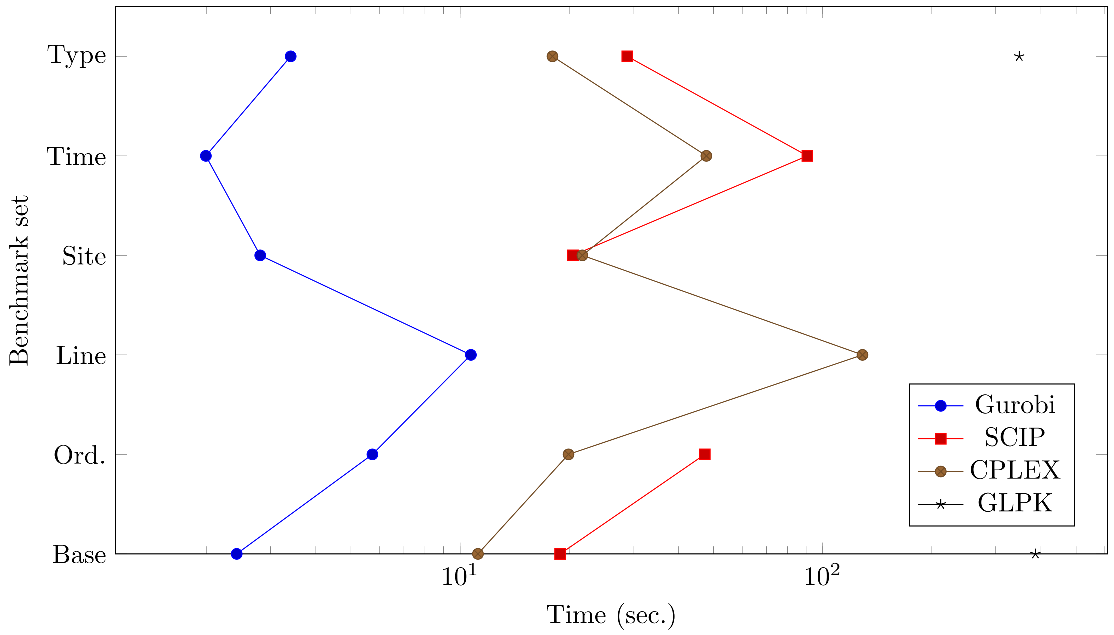


Figure 5.9: Average time needed to find a solution whose objective value is at most 50% away from the optimal solution on all test sets on the benchmark for doubling dimensions. Missing points for a set indicate a timeout on all instances of a set.

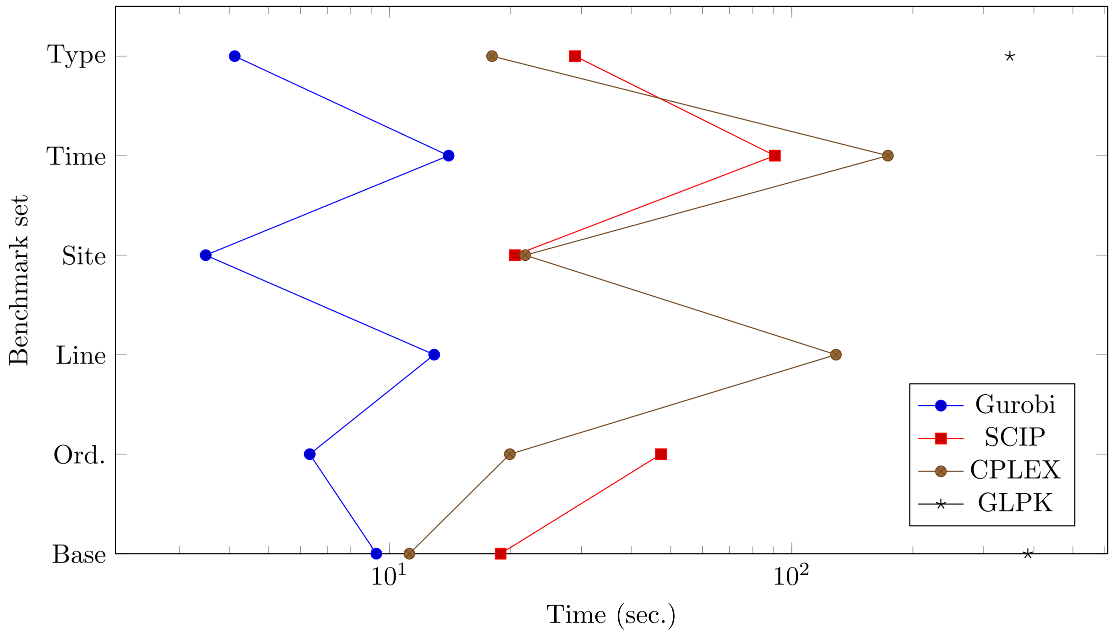


Figure 5.10: Average time needed to find a solution whose objective value is at most 10% away from the optimal solution on all test sets on the benchmark for doubling dimensions. Missing points for a set indicate a timeout on all instances of a set.

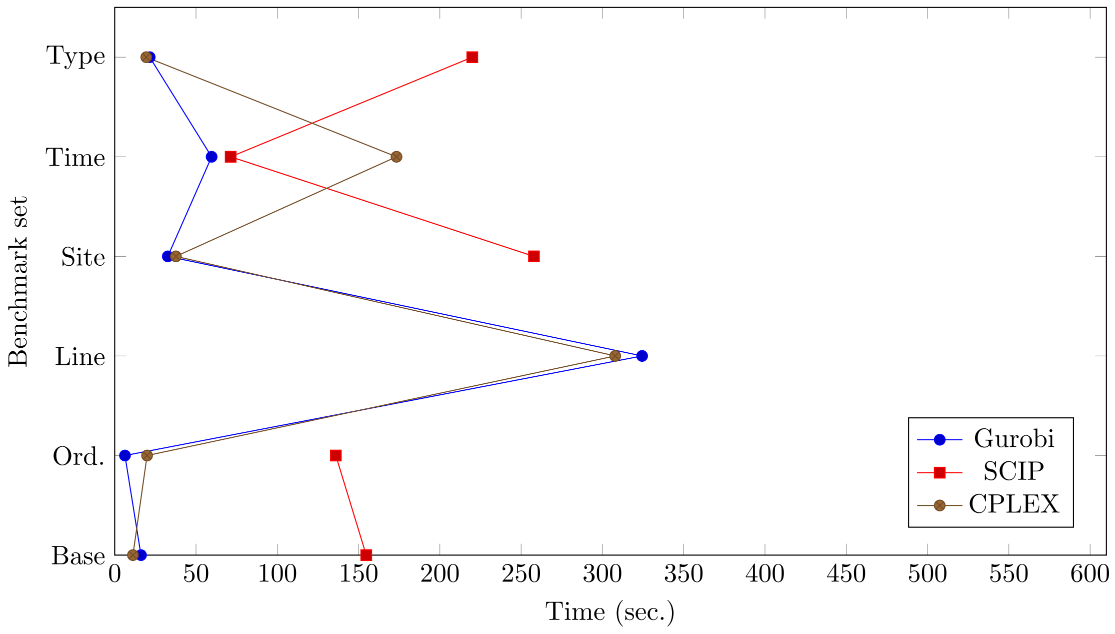


Figure 5.11: Average time needed to find a solution whose objective value is at most 2% away from the optimal solution on all test sets on the benchmark for doubling dimensions. Missing points for a set indicate a timeout on all instances of a set.

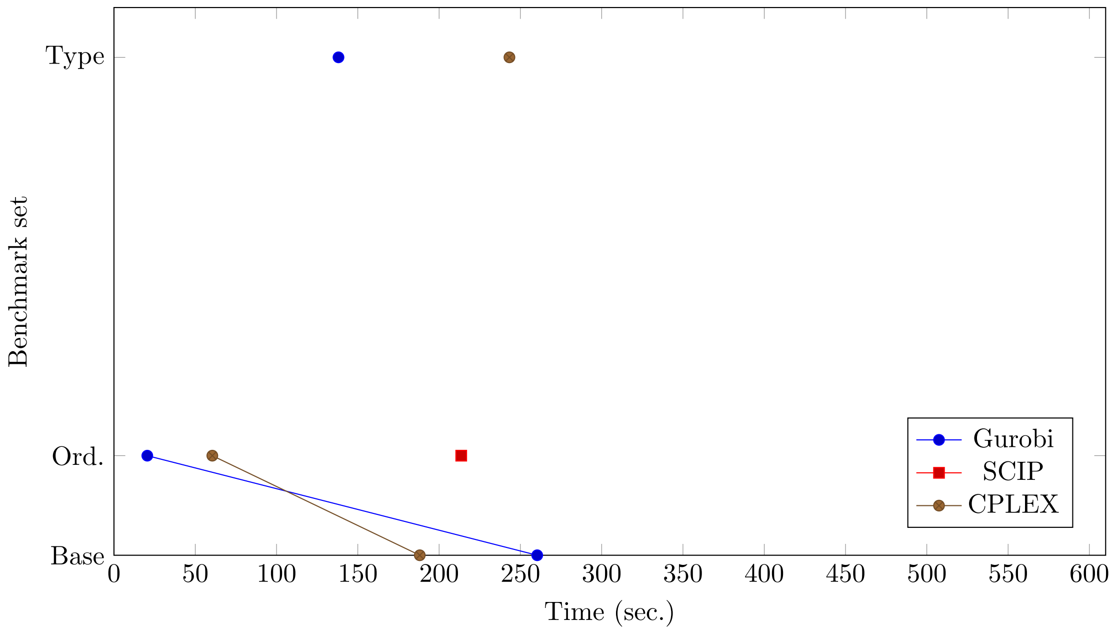


Figure 5.12: Average time needed to find a solution that is optimal on all test sets on the benchmark for doubling dimensions. Missing points for a set indicate a timeout on all instances of a set.

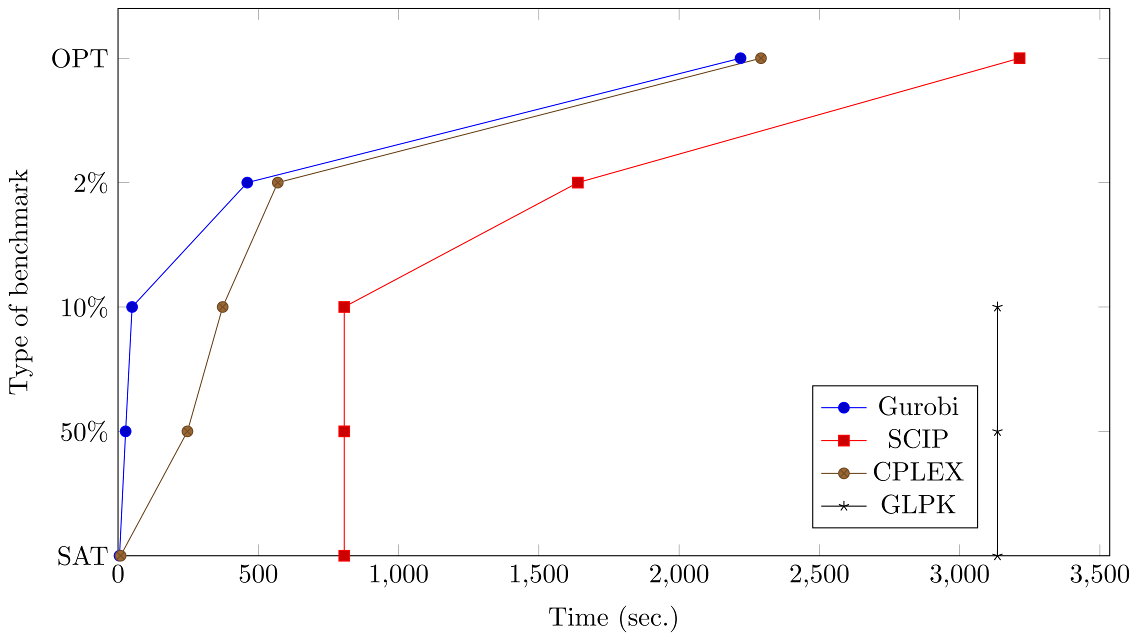


Figure 5.13: Average time needed for each MIP-solver to reach the given gaps on the benchmark for doubling dimensions. Missing points for a set indicate a timeout on all instances of a set. Time: Sum over all sets, taken from the "sum" entries of every benchmark table. Timeout runs weighted as 600 seconds, actual runtime is higher.

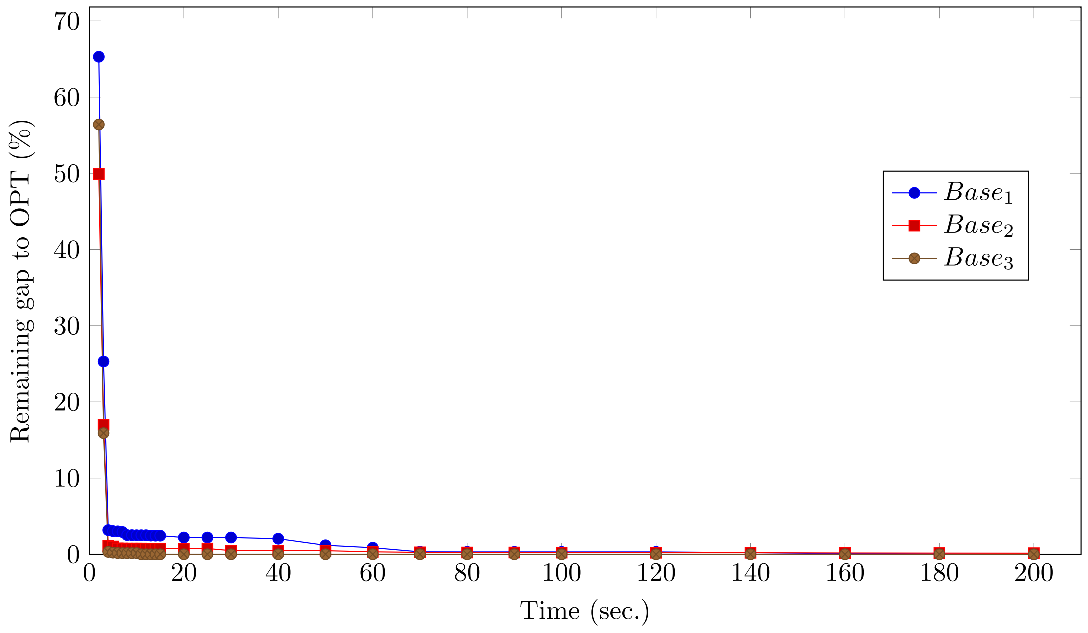


Figure 5.14: Reported solution quality and solution time of the solver Gurobi on the three seeded instances of the base benchmark set. The solver spends most of its time closing the last percent of the solution gap.

Name	Seeds	nOrders	nLines	nSites	nPeriods	nLineTypes
Set1	1,2,3	300	10	5	3	75
Set2	1,2,3	600	20	5	6	150
Set3	1,2,3	900	30	5	10	225

Figure 5.15: Complete list of all benchmark sets for our *converging dimensions* benchmark using artifical test data. Each set is defined by 3 differently seeded instances.

Testset	GUROBI	CPLEX	SCIP	GLPK	Z3	MSAT	CVC4
Set1	0.66	1.62	24.88	131.67	7.25	144.4	222.29
Set2	16.33	21.15	$1800^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	392.36	$1800^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Set3	200.68	124.01	$1800^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Sum	217.67	146.78	3624.88_{+}	3731.67_{+}	2199.61_{+}	3744.4_{+}	3822.29_{+}

Figure 5.16: Benchmark *converging dimensions*: Finding a single satisfying solution. Values measured in seconds. Runtime summed over all three instances of a set. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

periods and 300 line configurations. As discussed before, we are unable to test on the dimensions of the original problem due to insufficient memory. As we only have 5 sites in our original data and as we have seen that doubling the number of sites does not have a significant impact on the runtime with our previous benchmark, we chose not to scale this dimension.
The test sets are further detailed in the table of figure 5.15.
The results of our benchmark are given in the tables of figures 5.16 to 5.20. It is quite noticeable how this benchmark compares to our first one: The runtime for our second set is not much higher, and further underlines the observed runtime overhead by adding additional lines and time periods.

5.3.3 Optimizing Using SMT Solvers

As discussed in the preliminaries chapter, SMT solvers are inherently built to find single satisfying solutions. In practice, a valid solution of unknown quality is good to see whether the solution space is well defined, but may be unsuitable for actual planning. For this purpose, we implemented a routine that tries to incorporate a SMT solver into an iterative solving routine.

Testset	GUROBI		CPLEX		SCIP		GLPK	
Set1	1.37	4.11	1.69	5.08	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Set2	58.58	175.74	190.74	572.22	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Set3	$24.78^{\dagger\dagger}$	$1274.34^{\dagger\dagger}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$	$600^{\dagger\dagger\dagger}_{+}$	$1800^{\dagger\dagger\dagger}_{+}$
Sum	84.73_{+}	1454.19_{+}	792.43_{+}	2377.3_{+}	1215.8_{+}	3647.41_{+}	1243.89_{+}	3731.67_{+}

Figure 5.17: Benchmark *converging dimensions*: Finding a solution whose objective value is at most 50% away from the optimal solution. Columns: Runtime averaged over passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Testset	GUROBI		CPLEX		SCIP		GLPK	
Set1	1.37	4.11	1.69	5.08	15.8	47.41	43.89	131.67
Set2	58.58	175.74	190.74	572.22	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$
Set3	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$
Sum	659.95$_{+}$	1979.85$_{+}$	792.43$_{+}$	2377.3$_{+}$	1215.8$_{+}$	3647.41$_{+}$	1243.89$_{+}$	3731.67$_{+}$

Figure 5.18: Benchmark *converging dimensions*: Finding a solution whose objective value is at most 10% away from the optimal solution. Columns: Runtime averaged over passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Testset	GUROBI		CPLEX		SCIP		GLPK	
Set1	1.37	4.11	1.69	5.08	15.8	47.41	43.89	131.67
Set2	58.58	175.74	190.74	572.22	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$
Set3	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$
Sum	659.95$_{+}$	1979.85$_{+}$	792.43$_{+}$	2377.3$_{+}$	1215.8$_{+}$	3647.41$_{+}$	1243.89$_{+}$	3731.67$_{+}$

Figure 5.19: Benchmark *converging dimensions*: Finding a solution whose objective value is at most 2% away from the optimal solution. Columns: Runtime average over passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Testset	GUROBI		CPLEX		SCIP		GLPK	
Set1	260.35	781.05	188.11†	976.22$^{\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$
Set2	213.09†	1239.29$^{\dagger}_{+}$	152.03††	1656.09$^{\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$
Set3	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$	600$^{\dagger\dagger\dagger}_{+}$	1800$^{\dagger\dagger\dagger}_{+}$
Sum	1015.34$_{+}$	3046.04$_{+}$	1154.2$_{+}$	3462.6$_{+}$	1226.2$_{+}$	3679.53$_{+}$	1700.56$_{+}$	5101.68$_{+}$

Figure 5.20: Benchmark *convering dimensions*: Finding an optimal solution. Values measured in seconds. Columns: Runtime average for passed instances | Sum of runtimes over instances of set. Values measured in seconds, cut off after two decimal digits. Averaged over passed instances. Each † symbolizes a timeout on one instance of the set. Green entries: Lowest runtime on set.

Name	Seed	nOrders	nLines	nSites	nPeriods	nLineTypes
Inst1	1	10	5	5	3	20
Inst2	3	15	5	5	3	20

Figure 5.21: Test instances for demonstrating optimization using the Z3 SMT solver.

At any point during our routine, we want to be able to give an interval in which an optimal solution has to reside in. Iteratively, we try to find satisfying solutions in order to lower our upper bound, and simultaneously try to find unsatisfying solutions in order to raise our lower bound:

First, as in a regular run, a single satisfying solution is calculated, of which we can determine the objective value. We then start two workers in parallel: One with the added constraint that the objective value of a found solution has to be lower or equal than two thirds of our current solution value. Accordingly, a second worker is started that tries to find a solution valued at a third of our current solution value.
As soon as one of the workers finds a result, either satisfactory or unsatisfactory, both workers are terminated. If the found result is a valid solution, the upper bound of our current solution interval is updated to the value of the solvers added bound. Similarly, the lower bound is updated to the value below which a solution was to be searched for if the worker reported for there to be no solution. The idea behind this procedure is to do an aggressive search on the solution space in order to reduce the solution interval to a size such that the relative difference between the upper and the lower bound is below a certain percentage, similar to the so-called "gap" in MIP solving. In theory, one could extend this procedure to an arbitrary number of cores, splitting the interval accordingly. For this scenario, one may optionally not want to terminate all of the given workers at once, but possibly only those that were cut off by an intermediate result.

We did not have time to test our implemetation in detail, however, with a timeout of 1200 seconds, only very small test sets could be solved to a maximal gap of 5% using Z3, the fastest of our given SMT solvers. The sets we have tested are listed in the table of figure 5.21. A visualization of the development of the maximal deviation the optimal solution is given in figure 5.22 for the first instance and in 5.23 for the second instance. As one can observe, the solution time for both proving satisfiability as well as unsatisfiability increases the more we converge towards our optimal solution.

5.3.4 Runtime Behaviour on Real Data

We shortly want to go over the observed runtime behaviour of our implementation on real data at the Robert Bosch GmbH. Unfortunately, due to company and technical restrictions, we were unable to acquire precise numbers. Our runs were made using the solver CPLEX on a machine with a Xeon E5-1620 with 32 Gigabytes of RAM, running in parallel on 8 threads.

We were able to split the data into two sets that could be run independent of each other: The dimensions of the first set were around 800 orders, 20 lines, 5 production sites, 13 time periods and 50 line configurations, for which we were able to acquire a result with a gap of 10% after around 50 minutes, with a solution with a gap of 2% being found in the range of 2-3 hours. For a large number of orders in this set, an assignment to a line for the initial time period was already given and the variation in line configurations was relatively low, with a lot of lines having disjunct features, both present and upgradeable.
Our second set was smaller than the first one, with only around 300 orders, 20 lines, 4 production

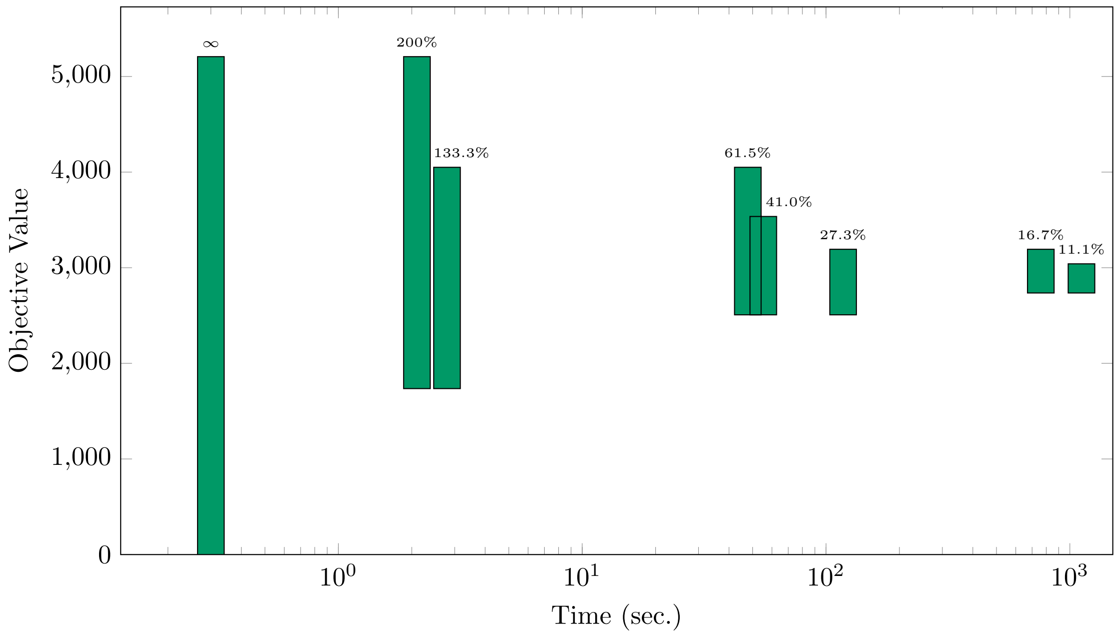


Figure 5.22: Gap delevopment of the solution value when using Z3 for optimization in our parallel solution loop on instance *Inst1*. Maximal gap: 5%. Every increase of the lower bound represents an UNSAT result, every decrease of the upper bound a SAT result.

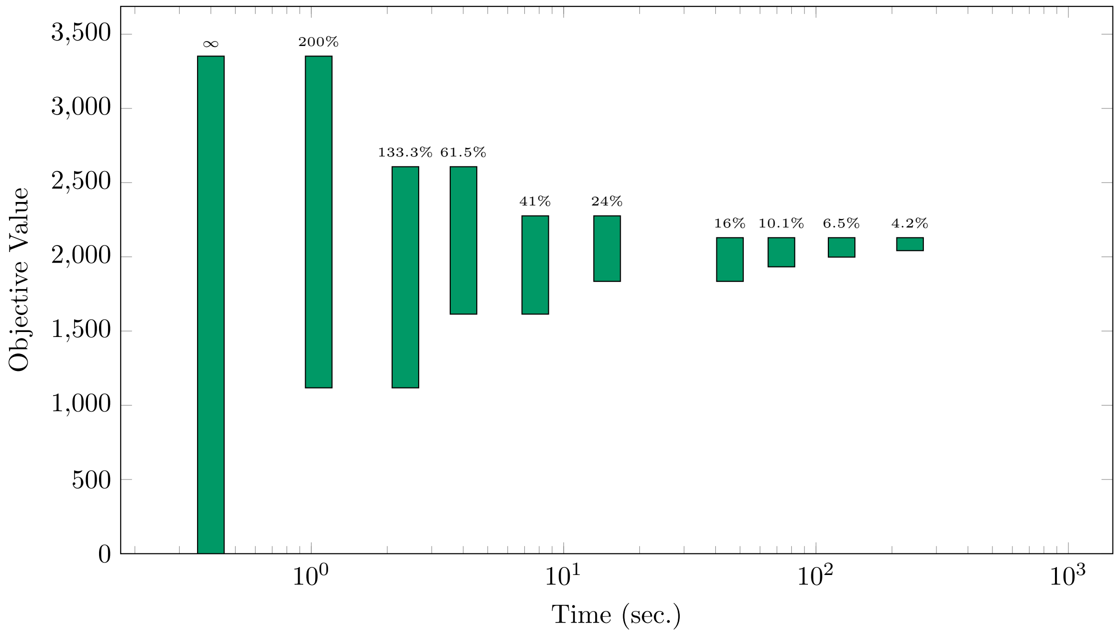


Figure 5.23: Gap delevopment of the solution value when using Z3 for optimizing in our parallel solution loop on instance *Inst2*. Maximal gap: 5%. Every increase of the lower bound represents an UNSAT result, every decrease of the upper bound a SAT result.

sites, 13 time periods and 64 line configurations. However, this set proved to be much harder to solve, with the solver only having found dual solutions but no valid primal solutions, even after 14 hours. This seemed curious to us as the dimensions of this problem were smaller than those of the first problem by a margin. As we allowed for new lines to be built in this scenario, we ruled out that the number of possible configurations was the relevant factor by reducing the number of possible line configurations to 16, with the runtime behaviour not changing. We suspect that the lack of initial assignments is responsible for the long runtime, however, we had no time to verify this assumption.

Chapter 6

Conclusion

In this work, we have solved an optimization problem by the Robert Bosch GmbH. We were to automate the planning process of assigning roughly 1100 orders to a set of production lines, while possibly upgrading or constructing new lines.
First, we developed compatible formulations for both satisfiability module theory (SMT) and mixed integer programming (MIP) solvers. For this purpose, we split the process into two parts: Constructing a model for the scenario in which lines cannot be upgraded and in which continious assignments of orders to lines are not possible and then constructing a complete formulation on the basis of this reduced model.
For the complete model we presented two alternative solutions. In the first one, the corresponding upgrade variable contained the information which configuration a line was upgraded from when doing an upgrade. This approach was rather easy to formulate, but proved to be too memory intensive for our implementation, as the number of these variables grew quadratically with the number of line configurations.
For our second approach, the upgrade variables no longer contained the infomation which type a production line was upgraded from, which resulted in additional constraints that we had to use as a workaround.

After the modelling process, we developed a software tool to extract the concrete data from files that were provided to us by the Robert Bosch GmbH. We implemented our models using the python libraries *Pyomo* and *pySMT* and used them to interface several different solvers, to then output a human-readable solution.

Finally, we benchmarked the runtime of our software and formulations on several benchmarks, investigating the runtime behaviour when doubling single dimensions of the problem, converging to the original problem dimensions and constructing a prototype to use SMT solvers for optimziation.

6.1 Future Work

While our model and implementation already yield plausible results, there are still some points that we would like to address for future work.
Our tool provides the user with all the necessary options to configure a number of parameters and to read and process data files, however only by the means of input via the command line. We recommend the implementation of a graphical user interface in order to better streamline the configuration of parameters and the selection of the input file.
Furthermore, although we have tried to use all the best practices during modelling that were

known to us, we deem it reasonable to invest additional time in trying to further improve our formulations.

Although being very memory-intensive, we recommend exploring the performance of our *compressed* formulation. If one can ensure that the number of given line configurations is very small ($\ll 100$), this formulation may be reasonable to use.

Bibliography

[1] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[2] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[4] J. P. Marques Silva and K. A. Sakallah. Grasp: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[5] R. J. Bayardo, Jr. and R. C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 203–208. AAAI Press, 1997.

[6] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency.* Springer, 2003.

[7] G. B. Dantzig. A history of scientific computing. chapter Origins of the Simplex Method, pages 141–151. ACM, New York, NY, USA, 1990.

[8] The Doxygen Documentation Software. `http://www.doxygen.nl/index.html`.

[9] The Pyxlsb Python Library. `https://pypi.org/project/pyxlsb/`.

[10] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[11] M. Gario and A. Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*, 2015.

[12] W. E. Hart, J. Watson, and D. L. Woodruff. Pyomo: modeling and solving mathematical programs in python. *Mathematical Programming Computation*, 3(3):219–260, 2011.

[13] W. E. Hart, C. D. Laird, J. Watson, D. L. Woodruff, G. A. Hackebeil, B. L. Nicholson, and J. D. Siirola. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, second edition, 2017.

[14] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[15] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, July 2018.

[16] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. ZIB-Report 18-26, Zuse Institute Berlin, July 2018.

[17] The CPLEX Optimization Suite. `http://www.cplex.com`.

[18] The GNU Linear Programming Kit. `https://www.gnu.org/software/glpk/`.

[19] The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`.

[20] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[21] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.