FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Solving Shift Design Problems with Answer Set Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael ABSEHER
Matrikelnummer 0828282

an der
Fakultät für Informatik
der Technischen Universität Wien

Betreuung:
Priv.-Doz. Dr. Stefan WOLTRAN
und
Priv.-Doz. Dr. Nysret MUSLIU

Wien, 31.01.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Solving Shift Design Problems with Answer Set Programming

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering & Internet Computing

by

Michael ABSEHER
Registration Number 0828282

to the
Faculty of Informatics
at the Vienna University of Technology

Advisors:
Priv.-Doz. Dr. Stefan WOLTRAN
and
Priv.-Doz. Dr. Nysret MUSLIU

Vienna, 31.01.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Michael ABSEHER
Walpersbach 198, 2822 Walpersbach

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I hereby want to thank my two supervisors, Stefan Woltran and Nysret Musliu, for their great support. Their encouragement as well as their expertise were invaluable for the progress of this thesis. Furthermore, I want to thank Martin Gebser and Torsten Schaub from the University of Potsdam for providing me with detailed insights in the world of Answer Set Programming during my visit in Germany.

At this point, I also want to express my gratitude to all the people from the DBAI group of the Institute of Information Systems, since it always was a pleasure to attend their lectures, respectively to work with them. Starting with Katrin Seyr, who invited me to work as a tutor at the DBAI group, I had the chance to meet a lot of experts at this working group and I could always feel like being at home. In this context, special thanks to Reinhard Pichler for being a great lecturer and for always supporting my requests.

I am deeply grateful to my parents for their guidance through all sections of my life and their patience during the last 24 years and I also want to pay a special tribute to my grandparents for all their support and encouragement. I will never forget my grandfather who left us after severe illness while this work was still in progress and I hope that he can be proud of what I have achieved in memory of him.

Finally, I want to thank my very best friend, Theresa Rasinger, for years full of fun and unforgettable moments. Without her, this work would not have been possible. Special thanks also to my friends Gerald, Jürgen, Bernd and Hans for attending many lectures with me, so that we had a good and successful time in very efficient teams. Last but not least, I want to express my gratitude to Nicole Wagner and Claudia Walla for their attempts to reduce my stress during the progress of this thesis.

Abstract

Shift design problems belong to a group of computational hard problems arising from economic needs for an efficient organization of a company's workforce. Given the demand of workers at each point in time, finding the best possible set of shifts and assigning the optimal amount of employees to each of the selected shifts is one of the most important tasks in the area of personnel planning.

To generate shift schedules in a comfortable and automated way, sophisticated exact and heuristic algorithms were developed in the last decades to tackle the various kinds of workforce scheduling problems. Apart from mathematical programming models, many of these algorithms are implemented in procedural or object-oriented programming languages. In this work, we will focus on declarative mechanisms in order to investigate their capabilities for solving real-life instances of shift design problems.

We propose three modelling approaches for the so-called "Minimum Shift Design Problem" which are implemented using the paradigm of Answer Set Programming (ASP), a declarative programming technique often described as the computational embodiment of non-monotonic reasoning based on the semantics of stable models.

Since ASP is able to investigate the whole search space in a structured way, it always finds the global optimal solution(s) in theory. In practice, this statement should indeed be treated with caution, since time is often the limiting factor. For this reason, we present a number of experiments and benchmarks in order to get an intuition of the performance of different solvers in combination with our programs.

Our experiments show that ASP performs well in many cases, although we have to admit that there is still work to do in order to obtain a competitive and robust tool for solving the Shift Design Problem, since the search space sometimes is too large to be handled efficiently by the exhaustive approach for search as implemented by ASP. Due to our encouraging results we are confident that we could provide a solid starting point for further research in the area of logic programming for solving optimization problems.

Kurzfassung

Aufgrund ökonomischer Überlegungen zum optimalen Einsatz des zur Verfügung stehenden Personals zählen Schichtplanungsprobleme bereits seit Langem zu den wichtigsten Punkte bei der Organisation vieler Unternehmen. Den Bedarf an benötigter Arbeitskraft immer möglichst exakt abzudecken kann einen entscheidenden Vorteil gegenüber Mitbewerbern bedeuten, aber auch für eine Vielzahl von gemeinnützigen Einrichtungen wie beispielsweise Krankenhäuser kann eine gute Auswahl der möglichen Schichten einen großen Effizienzgewinn bedeuten.

Aufgrund dieser immensen Wichtigkeit im Bereich der Unternehmensorganisation wurde bereits sehr früh versucht, den Schichtplanungsproblemen mit computergestützten Methoden zu begegnen um diese automatisiert und komfortabel lösen zu können.

Seitdem wurden zahlreiche exakte und heuristische Methoden entwickelt, um Werkzeuge für optimale Organisation zu liefern. Abgesehen von mathematischen Modellierungen wurden die meisten dieser Anwendungen unter Verwendung von prozeduralen sowie objektorientierten Programmiersprachen entwickelt. In dieser Arbeit werden wir unseren Fokus auf deklarative Konzepte legen und untersuchen, wie gut sich diese zur Lösung von Schichtplanungsprobleme einsetzen lassen.

Wir stellen drei Modellierungsansätze vor, welche Antwortmengenprogrammierung (ASP) als Programmierparadigma nutzen. ASP ist eine deklarativen Programmiertechnik, die oft auch als die Verkörperung des nicht-monotonen Schließens basierend auf der Semantik von stabilen Modellen bezeichnet wird. Das Ziel der von uns entwickelten logikorientierten Programme ist es, das sogenannte “Minimum Shift Design Problem” zu lösen.

Da ASP den kompletten Suchraum in strukturierter Art und Weise durchsucht, wird das globale Optimum, sofern existent, theoretisch immer gefunden. In der Praxis ist allerdings oft die Zeit ein limitierender Faktor, weshalb diese Aussage nicht als Garantie verstanden werden soll, dass es sich hierbei um die perfekte Lösung für unsere Problemstellung handelt. Um eine wissenschaftlich fundierte Aussage treffen zu können, präsentieren wir in unserer Arbeit eine Reihe von Experimenten, die es uns ermöglichen, ein Gefühl für die Leistungsfähigkeit unserer Programme in Kombination mit aktuellen Programmierumgebungen zu bekommen.

Unsere Ergebnisse zeigen, dass ASP in zahlreichen Fällen bereits jetzt vielversprechende Resultate liefert, es aber vermutlich noch ein weiter Weg ist bis Implementierungen entstehen, die ähnliche Leistungsfähigkeit aufweisen wie aktuellen Heuristiken. Aufgrund ermutigender Ergebnisse sind wir allerdings zuversichtlich, dass sich unsere Arbeit als ein solider Startpunkt für weitere Forschungen erweisen wird.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Shift design problems and its related problems are an important part of every day's life in most companies. These problems must fulfil hard and soft requirements concerning the management of employees or the utilization of production machines.

For an example scenario of a shift design problem, one can imagine an organization where we have to fulfil some requirements so that the number of assigned workers at any point in time is equal to the staff demand of the organizational unit. Additionally, the number of different shifts (for instance day and evening shifts with varying starting times and durations) should be minimal in order to reduce organization effort for constructing staff schedules.

Starting with the original set-covering formulation for the Shift Scheduling Problem [11], a lot of research has been done in the precincts of this problem and therefore, most of the problems are relatively well-studied. Despite this fact, computing the most desirable shifts in an automated way is still an interesting challenge for computer scientists all over the world.

Apart from mathematical programming approaches, like integer programming, many of the heuristic algorithms presented in the scientific literature are implemented using object-oriented or procedural programming languages (e.g. [14, 41] or [42]). This means that the programmer specifies exactly what the program has to do at which point of the process, in general consisting of a set of interacting procedures. This approach has several drawbacks, since the source code for these algorithms often contains thousands of lines of code which makes it hard to maintain and modifications often result in unreasonably high effort.

In this master's thesis, we focus on solving the so-called Minimum Shift Design Problem (MSD) [13, 34, 35] by means of Answer Set Programming (ASP) [23]. Briefly speaking, the goal of MSD is finding the minimal selection of shifts and the corresponding optimal amount of workers for these shifts so that the deviation from the actual demand is minimized. ASP itself is a declarative programming technique which is often described as computational embodiment of non-monotonic reasoning based on the semantics of stable models.

We investigate ASP, because we expect that it is possible to overcome the problem of rapidly decreasing maintainability and therefore using ASP might be worth a try, also in the area of optimization problems. Declarative techniques do not specify anything about *how* a given input has to be processed. Instead, they describe only *what* the expected result to be delivered in the presence of the respective input data should look like. These characteristics allow a much more convenient way of implementing a program and the resulting code can be adapted with very small effort, which makes the programs highly flexible.

Two prominent examples for declarative languages are for instance SQL, a popular query language that is used in the majority of current relational database systems and Prolog, a logical programming language with similarities to ASP. One drawback of declarative languages is the fact that often it is almost impossible to control the behaviour of the enclosing environment from within the source code. As an example, one mostly cannot specify the way, how a statement is processed and therefore, optimization can be a challenging task. Hence, this is an interesting question to investigate.

1.2 Aim of the Work

The main goals of our work are as follows:

- Modelling of real-world Shift Design Problems using Answer Set Programming

  To the best of our knowledge, currently no scientific investigations on implementations of the Shift Design Problem in ASP are known. Therefore, in this work we will study how challenging it is to model this problem in ASP. The aim is to present a solid starting point for other scientists to implement their own declarative programs tackling this and other workforce scheduling problems.

- Investigation of the performance characteristics of different ASP-environments

  In our work, we will compare the performance of two popular ASP environments by using our modelling approaches. For our experimental evaluation, the formula for the desirability of a solution is taken from existing literature concerning the Shift Design Problem in order to provide a high degree of comparability with existing results.

- Comparing our modelling approaches with state-of-the-art algorithms

  For our experiments we use existing problem instances that were used previously in the literature. In this work we will mainly focus on those data sets where the instances can be solved without deviation, but we also considered instances for which the optimal solution is not known.

For further comparison with other implementations, all instances that were used for this work were converted to a declarative input language similar to *smodels* [39], which is used by many ASP-environments. The data sets can be downloaded at the following location:

`http://www.dbai.tuwien.ac.at/proj/Rota/DataSetASP[1-4].zip`

1.3 Results of the Master's Thesis

The results obtained during our work on this thesis are as follows:

- We propose different modelling approaches for the Shift Design Problem using Answer Set Programming. We start with rather optimistic assumptions and a restricted search space in the first of our approaches and in the subsequent programs, this basic approach is extended so that we finally obtain a tool which enables us to deal also with more complex problem instances where no perfect solution exists, as this is the general case of the Shift Design Problem.
- Our modelling approaches were evaluated on the basis of benchmark instances inspired by realistic real-world situations. The results obtained during the experiments show, that ASP is very reliable for problem instances where a solution without deviation from the staff demand exists, but with an increasing number of optimization criteria, the performance often decreases significantly and so this could be a good starting point for further research and development.
- We found strong evidence that the performance of programs in the paradigm of ASP is heavily dependent on the actual ASP-environment used for the experiments, which is caused by the different implementations of the employed components.
- In our work, we explain all our approaches in detail and we also highlight some problems we came across during the implementation phase of our programs so that these pitfalls can be avoided in further developments. Additionally, we suggest some points for future work which shall provide a good starting point for other researchers.

1.4 Structure of the Master's Thesis

The remainder of this thesis is organized into the following chapters:

In Chapter 2 we will introduce the Shift Design Problem. This chapter of our work shall make the reader familiar with the basic concepts of workforce scheduling. Furthermore, we will give a detailed definition of the problem itself to provide a common vocabulary for the subsequent parts of our work.

In Chapter 3 we present the paradigm of ASP. This is done by first explaining the structural composition of an ASP-environment and the process steps from the program to the resulting answer sets. Afterwards we will introduce the basic terminology that is used to name the central parts of a program in ASP. Finally, the most important elements of the programming language that is used to implement our approaches will be described by means of practical examples.

In Chapter 4 we give a succinct overview of related work for the (Minimum) Shift Design Problem. Furthermore, we introduce other areas of application for the paradigm of ASP and we will have a look at some prominent software products that use ASP.

In Chapter 5 we present our three approaches to solve the Shift Design Problem. The first one requires that the problem instances can be solved without deviation and the assumed optimal count of shifts must be known already beforehand. The second program is quite similar, but at this time, the optimal number of shifts is computed automatically. Finally, the third approach can also deal with instances where a deviation from the requirements cannot be avoided.

In Chapter 6 we show the results that where obtained during the evaluation process. We start by first presenting the four data sets that were used. Afterwards, the experimental setting is documented. In the following parts of this chapter, the computational results are discussed and an analysis of our approaches is provided. The experiments aim at investigating the following characteristics of our approaches:

- Time needed to reach the best known solution
- Objective value obtained within a time bound

In Chapter 7 we summarize our work and give a perspective and some ideas for potential future work employing ASP as a powerful technique to solve problems in related areas of the Shift Design Problem.

CHAPTER 2

The Shift Design Problem

The Shift Design Problem is one of the sub-problems of workforce scheduling. The generic term *automatic workforce scheduling* in the context of computer science comprises problems that deal with arranging the workforce of a company or organizational unit in an optimal way with automated approaches.

In this chapter we will give an introduction to the basic concepts of workforce scheduling and we will also present a detailed definition of the Shift Design Problem we are going to solve in this work. In the subsequent explanations we adhere to the definitions and problem statements given in [34].

2.1 Basic Concepts of Workforce Scheduling

The desired outcome of workforce scheduling, the *workforce schedule*, can be described as a structured representation where the assignments of the employees to the available shifts for a period of time are recorded. In the literature, the term *workforce scheduling* appears with different names that can be used interchangeably. Labor shift scheduling, staff scheduling and rostering are only a small selection out of the large pool of synonyms.

Characteristics

The set of problem statements in the context of workforce scheduling often deals with common notions. Subsequently we will introduce these terms in order to provide a profound knowledge base for the reader to allow understanding the further parts of our work.

Although there are several different representations for workforce schedules, we focus on a definition that is almost perfectly suitable for our modelling approaches in the paradigm of ASP, but without loss of generality we could also have used a different representations as starting point for our subsequent implementations.

Shifts

Shifts are the central concept we will focus on in our work. Musliu [34] describes the term *shift* as period of time where a worker or a group of employees is on duty. Furthermore, a shift is often defined with a certain starting time and a specific length. Another way to define a shift can be by providing the ending time instead of the duration. For the following example schedules we will use the letters 'M', 'D' and 'E' as abbreviation for "morning shift", "day shift" and "evening shift". For example, the morning shift could start at 7 o'clock in the morning and have a duration of eight hours.

Indeed, in schedules appearing in practice there could be much more complicated structures. One example are additional breaks that could be necessary because of legal regulations. We will have a look at such extended problems in Chapter 4 which is presenting the related work.

Employees

As one can imagine, it is necessary for many organizational units to have a certain number of employees that can be assigned to the shifts in order to fulfil the staffing requirements. This number of workers is potentially arbitrary and solely depends on the organizational demands on which the respective problem instance is based. In general workforce scheduling problems it is possible that workers have different qualifications and therefore it is important to take these skills into account. Additionally, there are problem statements where different working times of employees have to be respected. The Shift Design Problem we investigate in this work does not deal with explicit employees so that we are not able to say which worker is assigned to which of the generated shifts. Instead, only the number of workers for each shift is part of the generated solutions.

Planning Period

The planning period or planning horizon describes the period of time between the start and the end of the schedule. In the following example and also for our evaluations we use one week as planning period. Other planning periods like schedules for one month are of course also possible, but the larger the chosen planning horizon is, the harder it gets for complete approaches like ASP to investigate the whole search space.

For our further explanations it is important to know that we can arrange similar parts of the planning period in groups without loss of generality. Often the planning horizon is divided into a number of days where every day consists of the same amount of time slots. Our approaches will use a distinction between the days of the week and each day is furthermore split into several time slots of equal length, for instance hours. This allows us to restrict the domain, the number of different values for a variable, and can lead to a significant improvement in terms of computation speed because of a more adequate way of defining the rules for our programs. One interesting natural effect of splitting up the planning horizon is that a shift could start at one day and end on the following day. In further definitions we will refer to these circumstances as *cyclic structure of the planning period*, which should not be confused with the term of cyclic schedules.

An Example Schedule

Table 2.1.1 illustrates a sample schedule for a complete week from Monday to Sunday where five employees are available to be assigned. Each row represents the schedule for exactly one employee. The columns of the table correspond to the day of the week and the content of the cells contains the information to which shift an employee is assigned. The symbol '-' in a cell is used to mark days off duty for the selected worker(s).

Employee	Mo	Tu	We	Th	Fr	Sa	Su
1	D	D	E	D	D	-	-
2	-	-	D	D	E	D	M
3	M	E	D	D	-	-	M
4	-	E	E	-	M	D	M
5	D	E	E	-	M	M	-

Table 2.1.1: A week schedule for five employees

When we interpret the information stored in Table 2.1.1 according to the scheme mentioned above, we can see for instance, that the employee with ID 1 works in the morning shift on Monday, Tuesday, Thursday and Friday. Additionally, we can state that he works in the evening shift on Wednesday and does not need to come to work on the weekend.

Workforce schedules can by divided into cyclic and acyclic ones. In cyclic schedules, every employee has the same skills with respect to the company's needs. Although there are some exceptional cases, this assumption is practical in the context of many workforce scheduling problems. One big advantage of rotating schedules is that for fairness reasons, the working plan for each person can be rotated within the group of workers. This means that at the beginning of the next week, the first employee can take the schedule of the second worker, the second employee can take the schedule of the third worker and so on. The last employee will then take the schedule of the first worker.

Employee	Mo	Tu	We	Th	Fr	Sa	Su
1	-	-	D	D	E	D	M
2	M	E	D	D	-	-	M
3	-	E	E	-	M	D	M
4	D	E	E	-	M	M	-
5	D	D	E	D	D	-	-

Table 2.1.2: Second week for the schedule from Table 2.1.1

Table 2.1.2 finally illustrates the second week for our cyclic example schedule. We can imagine that after five weeks the cycle is finished and a new one will start. In this way, rotating schedules can be used for instance to avoid complaints of employees which are not happy with their assigned plan, since every worker will have to use the same schedule in one of the next weeks.

Additional Constraints

Although it is possible to generate shifts and schedules without relying on constraints, it is very likely that this approach is not practical. To improve the practical usability, there is a variety of constraints that are applicable for the group of workforce scheduling problems. The most important ones will be discussed in subsequent parts of this section.

Temporal Requirements

One of the most important constraints in many workforce scheduling problems is the number of workers that is required to be present during each of the time slots within the planning period. In many cases it is not possible to satisfy the demand of workers perfectly, which means that some deviance is allowed. In the literature constraints with allowed deviation are called soft constraints. The goal with respect to soft constraints is that the deviation from a perfect solution is minimized. In the literature this is often described as optimizing a so-called *fitness value*.

The first two of our modelling approaches will treat the fulfillment of temporal requirements for workers as hard constraint. This means that any deviation is explicitly forbidden and only perfect solutions are accepted. The third program uses the fulfillment of temporal requirements as a soft constraint.

Subsequently we give an overview of some additional constraints which are important when the schedules are created. The number of workers required at each point in time is by the way the most important constraint we use in our modelling approaches for the Shift Design Problem.

Work and Rest Periods

Legal regulations in many countries specify that each employee has the right to have a specific number of days off after a period of working days. Therefore a suitable hard constraint for many workforce scheduling problems could be to limit the number of subsequent working days. Also it is maybe not desirable for the company leader that workers have too many days off in a row. In this case, a soft constraint could be used to minimize the probability of such schedules.

Sequences of Shifts

Due to ergonomic reasons it makes sense that some combinations of shifts in the schedule should be avoided. An example are sequences where a morning shift follows immediately after an evening shift, so that there is no adequate rest period for the assigned employee(s) between these consecutive shifts.

Average Working Time

To avoid complaints of workers about unfair organization of the schedule and in order to follow legal regulations, it can be required that the average number of working hours per planning period is around a specific value, for instance 38 hours. The goal is then to approximate this expected value as much as possible.

Individual Preferences

When all other constraints are satisfied to an acceptable degree, individual preferences of persons can be additionally taken into account. One of these preference could be that an employee wants to work primarily in night shifts. Indeed, this is possible only for non-rotating schedules.

2.2 Problem Statement of the Shift Design Problem

The original definition used in [34, 35] is given below:

Instance:

- n consecutive time intervals $[a_1, a_2), [a_2, a_3), \ldots [a_n, a_{n+1})$, all with the same length *slotlength* in minutes. Each interval $[a_i, a_{i+1})$ has an adjoined number of employees that should be present during that interval. Time point a_1 represents the begin of the planning period and time point a_n represents the end of the planning period.
- y shift types $v_1, \ldots, v_y$. Each shift type v_j has the following adjoined parameters: v_j.min_start, v_j.max_start which represent the earliest and latest start of the shift and v_j.min_length, v_j.max_length which represent the minimum and maximum length of the shift.
- An upper limit for the average number of working shifts per week per employee.

Problem:

Generate a set of k shifts $s_1, \ldots, s_k$. Each shift s_l has adjoined parameters s_l.start and s_l.length and must belong to one of the shift types. Additionally, each real shift s_p has adjoined parameters $s_p.w_i$, $\forall i \in \{1, \ldots, C\}$ (C represents number of days in the planning period) indicating the number of employees in shift s_p during the day i. The aim is to minimize the four components given below:

- Sum of the excesses of workers in each time interval during the planning period
- Sum of the shortages of workers in each time interval during the planning period
- Number of shifts k
- Distance of the average number of duties per week in case it is above a certain threshold.

Without loss of generality, we do not take the average number of duties per week into account in our work. We note, that this approach for reducing the complexity of the problem statement is also used in the existing literature, since this component of the optimization process is often considered as less important than the other ones. For instance, in [13] and [14] this criterion is omitted too. Subsequently, the formal representation of the original problem statement as defined in [34, 35] is provided:

Formal Definitions:

The generated shift belongs at least to one of the shift types if:

$$\forall l \in \{1, \ldots, k\} \exists j \in \{1, \ldots, y\} \mid$$
$$v_j.\text{min\_start} \leq s_l.\text{start} \leq v_j.\text{max\_start}$$
$$v_j.\text{min\_length} \leq s_l.\text{length} \leq v_j.\text{max\_length}$$

The sum of the shortages and excesses (in minutes) of
workers in each time interval during the planning period is defined as

$$ShortagesSum = \sum_{d=1}^{n} (Indicator(w_d - \sum_{p=1}^{k} x_{p,d}) * slotlength)$$
$$ExcessesSum = \sum_{d=1}^{n} ((Indicator - 1)(w_d - \sum_{p=1}^{k} x_{p,d}) * slotlength)$$

where

$$Indicator = \begin{cases} 1 & \text{if } w_d - \sum_{p=1}^{k} x_{p,d} \text{ is positive} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{p,d} = \begin{cases} s_p.w_i & \text{if time slot } d \text{ belongs to the interval of shift } s_p \text{ in the day } i \\ 0 & \text{otherwise} \end{cases}$$

The average number of working shifts per week per employee (AvD) is defined below:

$$AvD = \frac{(\sum_{i=1}^{k} \sum_{j=1}^{C} s_i.w_j) * AverageNumberOfHoursPerWeek}{\sum_{i=1}^{k} \sum_{j=1}^{C} s_i.w_j * s_i.length}$$

CHAPTER 3

Answer Set Programming (ASP)

In this work we will use the declarative approach of so-called *Answer Set Programming (ASP)* to compute solutions for the Shift Design Problem. Lifschitz outlines this programming paradigm with the following words: *Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. As an outgrowth of research on the use of nonmonotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications.* [31]

ASP is based on the stable model semantics of logic programming that was proposed by Gelfond and Lifschitz [23] and the process of finding solutions for a given problem instance in this paradigm boils down to compute stable models (answer sets). According to Lifschitz [31] the use of answer set solvers for search was identified as a new programming paradigm in [32] and [36]. Since then a lot of effort has been invested to improve the solvers in order to increase computation speed.

In the following sections we will introduce the basic methodology and terminology of ASP and we will also give an overview of the programming language that is used to implement our modelling approaches. Furthermore, we show the most important syntactical elements of the programming language we used by providing an example program. At the end of this chapter, we highlight differences of the two solvers which were used for our experiments.

3.1 Architecture

Two of the maybe most prominent examples for solver collections are the Potsdam Answer Set Solving Collection [19], a set of tools for ASP developed at the University of Potsdam as well as DLV [29], a deductive database system with various extensions developed at the University of Calabria in cooperation with the Vienna University of Technology. Regardless of which of those two environments one uses, the expressiveness of the solvers and their internal workflow that is executed in order to compute stable models is almost the same.

Programs in the paradigm of ASP have the significant advantage to be fully declarative and therefore the ordering of the rules in the program does not matter, which makes them highly flexible and maintainable while the overall effort for implementation and documentation shrinks appreciably. This modular and flexible structure of programs in the paradigm of ASP is also present in the solution process itself, since modern ASP-environments are mostly constructed according to the following workflow:

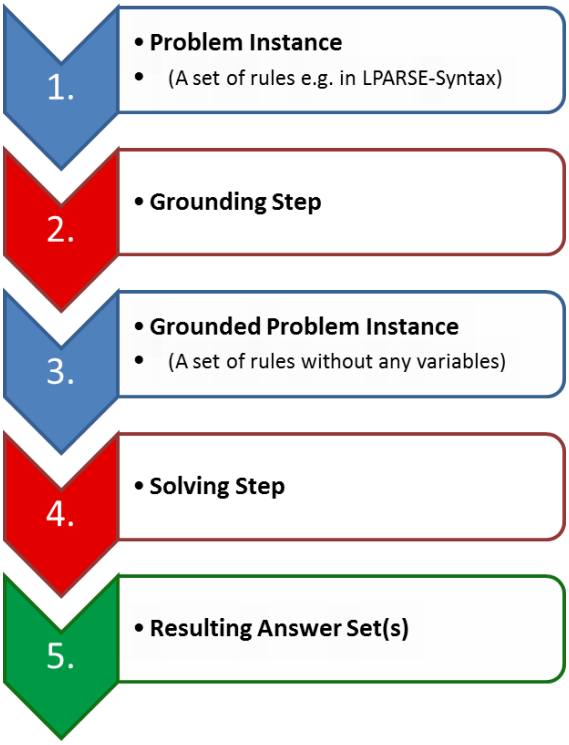


Figure 3.1.1: The process of ASP

Figure 3.1.1 shows the complete procedure of how a given problem instance is processed by the ASP-environment in order to obtain the resulting answer sets. At first, the problem instance is constructed by the programmer by using the input language of the ASP-environment. Afterwards the program is grounded. This means that variables within rules are replaced by their possible instantiations, so that we obtain a propositional (variable-free) program. The outcome of the grounding step is handled by the solving step in order to obtain a valid solution for the problem instance. The specific task of the solving step is to find those instantiations of rules so that all hard constraints can be fulfilled with the result that the generated answer set candidate is consistent.

3.2 Methodology

While there is no general way or specification for the process of interpreting and grounding arbitrary input programs, the structural approach of how logic programs in the paradigm of ASP are written is often very similar. This best-practice methodology is known as *GCO-scheme* [32]. It consists of three consecutive steps:

1. **G**uess:

   In this step of the computation process, the answer set candidates for the given problem instance are computed. This means that the search space is defined for the subsequent steps of the process.

2. **C**heck:

   The solution candidates generated in the previous step could violate some hard constraints and therefore they have to be removed. After the checking step, only consistent answer sets remain and they are delivered to the user or enquiring software component. The next step of the computation process helps us to deal with cases where we want to optimize some objective function(s).

3. **O**ptimize:

   This optional step allows to specify a measure of desirability for the answer sets that are obtained after removing invalid solution candidates. This measure is defined via a objective function so that every answer set is mapped to a natural number. By either minimizing or maximizing this objective function, the optimal answer set(s) can be found by the solver.

Sometimes the scheme of computing answer sets is described with different terms like for instance "Generate/Define/Test" [30], but the underlying workflow and its outcome is always the same. While this separation into a guessing step and a checking step can be used consequently in all ASP-environments, there are differences in the way how they deal with optimization. On the one hand there exist so-called weak constraints [9] as used by *DLV* and on the other hand there are weighted optimization statements as they are used in the Potsdam Answer Set Solving Collection. Although one has to use a different syntax for implementing optimization with these two alternatives, the semantics are exactly the same in order to provide the desired functionality for optimization rules. For more detailed information about the process of ASP, see for instance [29].

3.3 Terminology

By now we have only talked about the abstract word "rule" when we described the parts of a program in the paradigm of ASP. For the implementation section we will also make use of other specific terms that are important parts of the common vocabulary in the world of ASP.

Atoms and Literals

An atom is an expression $p(t_1, \ldots, t_n)$, where p is a predicate of arity n and $t_1, \ldots, t_n$ is a set of n terms. A classical literal l is either an atom p (in this case, it is positive), or a negated atom $\neg p$ (in this case, it is negative).

Positive or negative literals as mentioned above are only matched if it is explicitly known that the respective literal exists in the complete set of known literals, but often it is required to have some mechanisms to express the need that a literal must not necessarily exist. For these situations, negation as failure literals of the form $not\ l$ exist in most input languages. Such negation as failure literals are only matched in the case that l is not included in the set of known literals. This means that for instance $not\ l$ is assumed to hold unless l is derived.

Rules

Rules are the central components of programs in the paradigm of ASP. The position of the rules within the programs has no influence on the semantics, since ASP is fully declarative. Our implementations will use exploit this attribute to split the input problem instances from the production rules, so that no duplicate code is introduced. The general case of a rule is called *disjunctive rule* and the corresponding formula looks as follows, where h_1 to h_n as well as b_1 to b_n are literals, $n \geq 0$ and $m \geq k \geq 0$:

$$h_1 \vee \cdots \vee h_n \leftarrow b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_m$$

The disjunction $h_1 \vee \cdots \vee h_n$ is called the head of a rule r. The set of literals $b_1, \ldots, b_k$ is called the positive body and $not\ b_{k+1}, \ldots, not\ b_m$ is the negative body of r. A rule is matched if all the literals in the body are satisfied. New knowledge is then derived, since the head of the rule r must be true if the body of rule r evaluates to true.

Facts

Facts are rules without body. An empty rule body is always treated like the constant value $\top$ (verum), so that the corresponding formula in the head of the rule has to be satisfied in every answer set. For convenience reasons, the symbol "$\leftarrow$" is often omitted in the programmatic declaration of facts.

Constraints

If the head of a rule r is empty, by definition the rule is extended with an implicit atom a which occurs in its positive form in the head and preceded by *not* in the body of the original rule. For instance, the rule r given by $\leftarrow b$ is identical to $a \leftarrow not\ a, b$. Therefore, the program containing r has no answer sets in case that b is known to hold, because the rule r can never be satisfied based on the fact that deriving a and $not\ a$ at the same time is impossible. Therefore rules with empty head are called *(integrity) constraints*. Walsh [43] investigates how constraints can be exploited in order to improve the efficiency of a program not only for ASP, but also for integer linear programming, propositional satisfiability and pseudo-boolean solving.

3.4 Semantics

In this section, we provide an overview of fundamental terms concerning the semantics of stable models which are important for a deeper understanding of ASP. We start with an introduction to the theoretical backgrounds of ASP and afterwards, we focus on the actual process of how answer sets are computed.

Definition 3.4.1 (Language of a Logic Program):
The language $\mathcal{L}_\Pi$ of a logic program Π is defined by the constants, variables, function and predicate symbols (with their corresponding arities) occurring in Π. If Π contains no constants, an arbitrary constant c is introduced.

Definition 3.4.2 (Herbrand Universe):
Given a logic program Π, the *Herbrand universe* of Π, denoted by $\mathcal{U}_\Pi$, is defined as the set of all ground (variable-free) terms which are occurring in Π. In case that there are no ground terms present in the program Π, an arbitrary constant is introduced as single element in $\mathcal{U}_\Pi$ in order to act as a dummy element for further steps. Informally, the Herbrand universe of a given first-order language $\mathcal{L}$ can be interpreted as the set of all ground terms which can be formed with the functions and constants in $\mathcal{L}$.

Definition 3.4.3 (Herbrand Base):
The *Herbrand base* of a logic program Π, denoted by $\mathcal{B}_\Pi$, is the set of all ground atoms which can be formed by using the terms in $\mathcal{U}_\Pi$ in combination with the set of predicates defined in the program Π. In other words, the Herbrand base of a first-order language $\mathcal{L}$ is the set of all ground atoms which can be formed with the functions, constants and predicates in $\mathcal{L}$.

After we have defined the language, the Herbrand universe as well as the Herbrand base of a logic program, we will now have a look at their application in the context of ASP. In order to compute answer sets more easily, grounding is used to obtain a variable-free representation of a given program. Note that this step can have significant impact on the performance of the program implemented in ASP, depending on the number and domain of the variables, since any feasible replacement has to be considered.

Definition 3.4.4 (Grounding):
The grounding $ground(r)$ of a rule $r \in \Pi$ is the set of rules obtained by replacing the variables occurring in r with all elements in $\mathcal{U}_\Pi$. The grounding of the whole program Π, denoted by $ground(\Pi)$, is defined as the set union of the groundings for all rules in Π.

The final outcome of the grounding task is a program with equal semantics as the original one, but it no longer contains any variables. Afterwards, the following definitions can be used to compute the actual stable models, respectively the answer sets, for a program in the paradigm of ASP. The definitions provided below also explain why the step of grounding can be executed separately from the process of solving the problem instance, as depicted in Figure 3.1.1.

Definition 3.4.5 (Herbrand Interpretation):
A *Herbrand interpretation* I of a logic program Π is a set of atoms from its Herbrand base $\mathcal{B}_\Pi$. A rule $r \in \Pi$ is satisfied by I, if $h(r) \cap I \neq \emptyset$ or if the body fulfils the following criteria:

$b^+(r) \setminus I \neq \emptyset$ (Not all parts of the positive body of r are element of I.)
$b^-(r) \cap I \neq \emptyset$ (Some part of the negative body of r is an element of I.)

Definition 3.4.6 (Herbrand Model and Answer Set):
A Herbrand interpretation I of a logic program Π is a *Herbrand model* of Π, if it satisfies every rule in Π. Furthermore, I is called *answer set* in the case that it is a subset-minimal model of the so-called Gelfond-Lifschitz reduct of Π with respect to I, defined as follows:

Definition 3.4.7 (Gelfond-Lifschitz Reduct):
The *Gelfond-Lifschitz reduct* of a logic program Π with respect to a Herbrand interpretation I is defined as the logic program that is obtained by applying the following replacement:

$$\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$$

Definition 3.4.8 (Answer Sets of Non-Ground Logic Programs):
An Herbrand model I of a logic program Π is an answer set of Π if I is at the same time also an answer set of the grounding of the original logic program Π, defined by $ground(\Pi)$.

3.5 Programming Environment

For the implementation of our modelling approaches, we used the Potsdam Answer Set Solving Collection as development environment. Especially, we used the grounder *gringo* [21] to convert the rules of the program to a format that can be handled by the solvers *clasp* [22], respectively *unclasp* [1]. In this section we will give a short overview of the most important elements of the language we used to implement our programs and highlight the differences of the two solvers used during our investigations.

The input language of *gringo* is similar to the language of *lparse* [39], but there are some differences between the two languages. For instance, *gringo* provides full support for variables within compound terms and also some additional aggregates which can reduce the effort needed for the development of new answer set programs. More information about the third release of the grounder *gringo* can be found for example in [21]. This software release of the grounder was also used for our implementations during the evaluation.

The following program acts as an example for a simple logic program in the paradigm of ASP, implemented using the tools from the Potsdam Answer Set Solving collection. Note that the character combination `:-` is used by most ASP-environments as the syntactical analogy to the symbol $\leftarrow$. Furthermore, the declaration of a rule in the input language of *gringo* rules is terminated by a full stop.

```
1  node(1).
2  node(2).
3  node(3).
4  edge(1,2).
5  edge(1,3).
6  edge(2,3).
7  color(r).
8  color(g).
9  color(b).

10 1 { nodecolor(X,C) : color(C) } 1 :- node.

11 :- edge(X,Y), nodecolor(X,C), nodecolor(Y,C).
```

Program 3.5.1: 3-Colorability

Program 3.5.1 represents one possible ASP-program for the 3-colorability problem. The goal of is to find a valid assignment of colors to the nodes of a graph where no adjacent nodes share the same color. In lines 1 to 9 of the program code, we can see the knowledge base of our program, consisting of facts representing the nodes and edges of the graph as well as the available colors.

Programs containing rules with disjunction in their head formula have a significantly higher computational complexity than those programs where no disjunctions occur [16]. Therefore *gringo* allows an alternative way to define feasible rules for the guessing step of ASP.

Predicate `nodecolor` in line 10 is used to choose exactly one color for each node in the graph, since the lower bound (on the left-hand side of the curly brackets) and upper bound (on the opposite side) for the selected literals are both set to the value 1. Another way to implement disjunctions in an efficient way in *gringo* is to use the same syntax as above, but with square brackets instead of curly ones. By doing so, one can specify an upper and lower bound for the aggregated sum of a mathematical formula that is applied to the literals within the brackets. In line 11, we finally define a constraint which is used to ensure that for no two adjacent nodes the same color was chosen.

3.6 Differences between *clasp* and *unclasp*

The main difference between the solvers *clasp* [22] and *unclasp* [1] is the approach how they deal with optimization criteria. While *clasp* successively refines upper bounds which are witnessed by solutions, the unsatisfiability-based approach of *unclasp* relies on the iterative extraction and relaxation of (local) inconsistencies in view of too tight bounds for the optimization criteria. In simplified words, *clasp* starts with upper bounds while *unclasp* starts from lower bounds, causing *clasp* to generate suboptimal answer sets in order to refine the internal bounds.

Given that aforementioned inconsistencies implying the infeasibility of bounds under which they are encountered, the first solution discovered by *unclasp* after performing all the necessary relaxations is guaranteed to be optimal. In the subsequent parts of our work we will see, that depending on the structure of the programs, both approaches have advantages and disadvantages for solving the Shift Design Problem.

CHAPTER 4

Related Work

In this section we will give an overview of related work in the area of shift scheduling and highlight some relevant ASP-implementations. Although ASP is a well suited tool for declarative problem solving in the area of knowledge representation and reasoning, it has not yet attracted broad attention in other research areas covering optimization problems. Nevertheless, in the second part of this chapter we will highlight some of the most prominent examples where ASP was employed to solve complex problems.

4.1 The Shift Design Problem and Related Problems

As mentioned in Chapter 2, the Shift Design Problem is only one example out of the variety of problems that are related to workforce scheduling. In this section we will have a look at shift scheduling as well as other related problems and provide references to scientific literature in order to allow a better understanding of this important group of economically inspired problems.

The Shift Design Problem we consider in this thesis is relatively similar to the so-called *Shift Scheduling Problem*. The original set-covering formulation of this problem was introduced by Dantzig [11]. In this approach, the available shifts are enumerated based on their starting points in time, their duration and also breaks as well as break windows are taken into account. Since the exhaustive enumeration of all feasible shifts soon leads to a rapid increase in the size of the search space, the method proposed by Dantzig becomes less practical when the number of shifts increases. This is due to the fact that each new instance of a shift goes hand in hand with introducing a new variable representing the respective shift. To overcome this problem, Bechthold and Jacobs [6] presented an integer programming formulation of the Shift Scheduling Problem where breaks are modelled implicitly in order to decrease the computational workload compared to the original problem statement.

A early experimental evaluation based on a set of real-world problem instances of the Shift Scheduling Problem is provided by Henderson and Berry [26]. In their work, the performance of two heuristics for the shift scheduling of telephone operators was investigated.

Another model for the Shift Scheduling Problem was developed by Thompson [40]. This integer programming approach combines the work of Moondra [33] and Bechtold and Jacobs [6] with the goal that the lengths of the shifts as well as breaks can be modelled implicitly.

Aykin [2] presented an integer programming approach for modelling the Shift Scheduling Problem with flexible ways to define rest and lunch breaks with multiple break windows that overcomes some of the limitations of [6] so that also shifts with durations of more than 24 hours can be modelled. In recent literature, Côté et al. [10] proposed an grammar-based approach for solving the multi-activity shift scheduling problem, where context-free grammars are used to generate an integer programming model of the respective problem instance. A detailed comparison of different solution approaches for the Shift Scheduling Problem can be found in [3].

Although there are many similarities between shift scheduling and the (Minimum) Shift Design Problem, there are some important differences. One of them is the fact that the latter deals with multiple days in general while shift scheduling is supposed to generate schedules for a single day. Another distinguishing characteristic is that under-staffing is allowed in the definition of the Shift Design Problem, while it is forbidden in the original formulation of the shift scheduling problem. It is assumed that the Shift Design Problem is more difficult than shift scheduling, since the selection of the shifts must be handled more carefully to obtain an optimal solution for a given problem instance with multiple days: *Scheduling a full week adds the difficulty of having to reuse shifts on all days of the week in order to minimize the overall number of shifts used. In addition, the weekly problem is cyclic, so special care must be taken to connect the end of the schedule to its beginning.* [35, p. 4]

The original definition of the Shift Design Problem was presented in [34, 35] and further investigations on the problem are provided for instance in [18]. In order to improve the quality of the solutions and computational efficiency, hybrid approaches were developed. Those hybrid solvers combine two or more heuristics. A hybrid heuristic is used by Di Gaspero et al. [13] to compute solutions for the (Minimum) Shift Design Problem. Their hybrid approach combines a greedy heuristic with a local search algorithm. A hybrid approach combining local search and constraint programming for solving instances of the Shifts and Breaks Design Problem can be found in [14].

Further examples for papers that contain investigations on local search algorithms in the broad area of workforce scheduling are [8, 41, 42]. Constraint programming techniques were used by Lau and Lua [28] to compute schedules for television personnel with respect to the skills of each crew member. Note that there exists a variety of different related problems and variants of these problems. For more information, we refer the reader to existing surveys on workforce scheduling, like [17] and [12].

4.2 Applications based on ASP

ASP has been used to solve computational hard problems in many application contexts like product configuration [38], synthesis of multi-processor systems [27], decision support for space shuttles [4, 37] and many more. Baral [5] for instance also mentions network management, social modelling, security engineering, multi-agent systems as well as compiler optimization as the additional areas of application of ASP, but because of the intuitive syntax and semantics of this paradigm, this listing is indeed not exhaustive.

For example, one of the main strengths of ASP is in the domain of planning problems. An introduction of planning can be realized in ASP can be found in [15]. In this paper written by Dimopoulos et al., the SMODELS-environment [39] was used to implement a framework for planning. Maybe one of the most famous products is the so-called "USA-Advisor" [4], a software that was used to help the NASA flight controllers by providing decision support.

As already mentioned in the introduction of our work, real-world staff scheduling problems are often handled with heuristics that are implemented with procedural programming languages like C or C++. During our literature research we could not find any scientific implementation to solve real-world Shift Design Problems by employing ASP.

Although we could not find any comparable programs for designing shifts, there exists a prominent implementation in ASP for team-building problems: Grasso et al. [24] implemented an application in the paradigm of ASP with the purpose to support the process of team-building at the Gioia-Tauro seaport in Italy. The main goal of this program is to assign the right group of employees according to their skills to every arriving and departing cargo boats in order to load and unload their cargo as fast as possible.

To complete the chapter about related work, a detailed overview of ASP and its relation to other programming paradigms is given in [7]. A survey about commercial and also some free software products based on ASP can be found in [25]. Gebser et al. [20] finally provide a compact overview of current challenges of ASP and try to give an outlook on future steps that could be taken in order to promote ASP in areas of application which are relatively unexploited by declarative languages at the moment.

CHAPTER 5

Solving the Shift Design Problem

In this chapter of our work we will elucidate the structure of the implementations fitting to the three modelling approaches that were selected to be presented in our work. The goal of this part of the thesis is to discuss the ideas behind the code fragments of our programs in an intuitive way for the reader.

We will start by giving an introductory example in Section 5.1 that will be used to explain the rules of our approaches step by step in the remaining parts of this chapter. The problem instance was constructed in such a way that it is solvable without shortage and excess and moreover, the example has only one optimal solution, so that the reader can easily convince himself that the idea behind our programs is correct.

Afterwards, in Section 5.2, we will have a closer look at the common knowledge base for all three programs. The general term *knowledge base* in principle denotes the whole set of facts and constants that act as input values for our modelling approaches. In Section 5.3, we will then present the complete ASP-representation of the simplest method. In this implementation of the Shift Design Problem we search for an exact solution and rely on the fact that the optimal count of different shifts is already known beforehand, so that there is no further optimization needed and only the satisfiability of the problem instance with a given number of shifts has to be checked. In the following Section 5.4 we will inspect the extension of the first modelling approach where the solution still has to be exact, but the optimal shift count is not known in advance any more. Finally, in Section 5.5 we focus on a flexible representation that enables us also to solve problem instances where no exact solution exists.

To improve clarity, we will go step-by-step through all parts of our implemented programs and explain their functionality as well as their impact on the search for the optimal solution in detail.

5.1 A Small Example

On the following two pages of our work, we will present an introductory example for the Shift Design Problem that will enable us to fully understand the process of finding an optimal solution with the programs we have implemented.

Since this simple instance was created solely for the purpose of explaining the general idea behind the respective predicates that form our programs, we have restricted the planning period to a single day with eight time slots to keep the size of the resulting answer set compact.

Table 5.1.1 defines the allowed starting points in time for three shift types. The second column of the table holds the zero-based index of the time slot where the selected shift can be started and the following two columns specify the maximum allowed deviance in full time slots from the default starting time.

To clarify the information stored in the table, we can pick for instance shift type 2: By default, it's planned that the shift starts on the fifth time slot, but an employee is also allowed to begin one time slot earlier, because the maximum allowed negative deviance is specified with the value 1. The column for the allowed positive deviance of shift type 2 contains the value 0, so that we can conclude that an instance of the second shift type must not start later than the time slot with index 4. With the same approach we can now also specify all possible instances of the two remaining shift types, but for the sake of brevity we will skip these redundant explanations.

Shift	Start	Positive Deviance	Negative Deviance
1	2	0	0
2	4	0	1
3	6	1	0

Table 5.1.1: Possible shift starts

Until now we don't know anything about the possible shift durations, so Table 5.1.2 contains this remaining information needed to completely specify our shift types. This second table can be interpreted in a similar way than Table 5.1.1 and after combining the information stored in these two tables we have full knowledge about all possible shifts that are available in our problem instance.

Shift	Length	Positive Deviance	Negative Deviance
1	3	1	1
2	3	1	1
3	3	1	1

Table 5.1.2: Possible shift lengths

The only remaining information we still need to complete our introductory example is the demand of workers needed at each of the eight timeslots. The trend of requirements is depicted in Figure 5.1.1. The columns of the grid correlate to the respective time slot and the greyish cells represent the number of workers that are required at each point in time. For instance we can see that we need exactly one employee at time slots 0 and the number of workers needed at the time slot with index 7 is three.

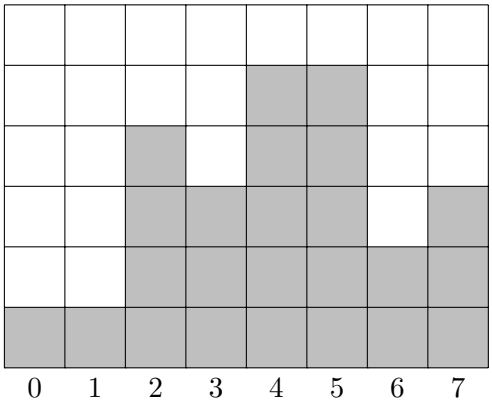


Figure 5.1.1: Initial situation of introductory example

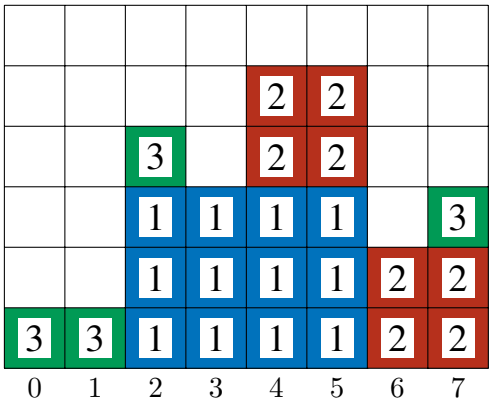


Figure 5.1.2: Solution for the introductory example

Figure 5.1.2 shows the one and only optimal solution for our problem instance. The numbers in the cells as well as the different colors denote the selected shifts that actually contribute to the coverage of the respective requirements. For instance we can see that we have selected an instance of shift 3 with one worker starting from the time slot with index 7 and a duration of four time slots. By means of this sample shift 3 we can also highlight the cyclic structure of the planning horizon, since the remaining three time slots of duration that cannot be used in the originating cycle are taken into account in the next one. During the following explanations of the rules forming our programs we will show how this optimal solution was generated by each of our programs in more detail.

5.2 Common Knowledge Base

In this section, we will present the general structure and meaning of the constants and facts which are used as input data for all three modelling approaches. This information differs from problem instance to problem instance, while the code fragments in Section 5.3, Section 5.4 and Section 5.5 do not need to be modified. To keep the code highly maintainable and flexible, the input is therefore always strictly separated from the productive program elements and kept in individual files.

Constant `days`

```
1 #const days = <number of days>.
```

Constant 5.2.1: `days`

The constant `days` is used to set the number of days which is used as planning period. The planning period is always cyclic, this means that the shifts fit seamlessly together after the specified amount of days. For instance, one can imagine an evening shift starting on Sunday at 10 pm with a duration of eight hours. Clearly, a worker who is assigned to that shift has to stay until 6am on Monday and cannot go home after just two hours of work, although the week is over at Sunday midnight.

Constant `timeslots_per_day`

```
1 #const timeslots_per_day = <number of timeslots per day>.
```

Constant 5.2.2: `timeslots_per_day`

The constant `timeslots_per_day` defines the precision of the assignments by setting the timeslots that are available each day of the planning period. Together with the constant `days`, the value of `timeslots_per_day` acts as configuration skeleton for the following predicates used in the input files.

Additionally, a small modification to this value can have big influence on the program's performance when employing Answer Set Programming as solution method, since the number of answer set candidates that need to be checked is heavily dependent on the number of timeslots per day.

We will now give an explanation why this value is really important for the performance. Imagine a shift that has a duration of at least eight hours and the employees are allowed to do at most one hour of overtime. When `timeslots_per_day` is set to 24, then every time slot represents one hour and we have exactly two possibilities to create a valid shift. The first option is to work eight time slots, respectively hours, and the other one is to do one time slot of overtime.

When we now double the precision to 48 time slots per day, we could also work 30 minutes longer than required and therefore there is a third possibility to create a valid shift. By again doubling the amount of available time slots per day, we already obtain five solutions, since the two 30-minute-slots of overtime can be split into two 15-minute-slots each.

To make the simple example from above more convincing, we can extend it towards flexible starting times and durations, so that an employee can choose the starting time and the shift lengths more freely. For instance starting between 7am and 9am with a shift duration between seven and nine hours. In this case, one can easily assert that the number of possibilities grows very rapidly. To model this challenge in an appropriate way, we introduce the knowledge of these flexible shift settings in the two upcoming predicates `shift_start` and `shift_length`.

Predicate `shift_start`

```
1 shift_start(<type>, <start>,
2             <allowed_positive_deviance>,
3             <allowed_negative_deviance>).
```

Predicate 5.2.1: `shift_start`

The predicate `shift_start` is used to define the start of a respective shift. The type of the respective shift is hereby defined via the attribute `type`. In the problem instances used in our evaluations, the information about the type is represented by the abbreviations `m` (morning), `d` (day), `e` (evening) and `n` (night).

The type can also be seen as name for the shift and it plays a central role when merging `shift_start` with the following predicate `shift_length`. The way, how these two very important input facts work together will be described in a moment.

The actual starting time slot of the shift is set by the argument `start`. The value has to be in the range from 0 to `timeslots_per_day` - 1. The programs expand this information to all days in the planning period, so the user does not have to specify the same slot again for each day. Via the last two arguments of the predicate, the program's operator can define allowed deviances. A positive (negative) deviance, for instance of value 1, would allow an employee to start his working time one time slot later (earlier) than specified by the argument `start`. Like all other numeric input values in the programs, the two arguments `allowed_positive_deviance` and `allowed_positive_deviance` have to be greater than zero.

To increase the comfort for a user, it is not necessary to define a shift twice, if the variation would lead to a negative starting time or a value greater or equal to the time slots per day. In this case, the programs will automatically distinguish between the current day and the adjacent one and thus ensure correct handling of this issue.

When remembering our introductory example, we can see that we can immediately transfer Table 5.1.1 to instantiations of the predicate `shift_start`. The same also holds for the following input predicates of the knowledge base of our three proposed programs.

Predicate `shift_length`

```
shift_length(<type>, <length>,
             <allowed_positive_deviance>,
             <allowed_negative_deviance>).
```

Predicate 5.2.2: `shift_length`

After the starting times are set, the predicate `shift_length` is used to define the duration of each of the shifts. The type of the shift is set via the argument `type`, like before. The amount of time slots that an employee has to work in the respective shift is configured via the value `length`. To allow overtime (positive deviance) or leaving work earlier (negative deviance), the operator can again specify the appropriate value to a non-negative integer of his choice.

Association between `shift_start` and `shift_length`

The predicates `shift_start` and `shift_length` are strongly connected with each other, since combined they are used to define the complete information about all possible shifts. The mentioned predicates allow passing necessary input information about all the different shifting possibilities to our programs in an easy and intuitive way for an end-user.

Both types of facts are connected via the common argument `type`, so that the responsible ASP-environment can put the instantiations of the predicates in the correct context. A reader now may ask why we do not merge the two predicates into one with an arity of seven that does not require any consolidation step.

There are two reasons for us to choose this modelling approach. At first, it is best practice in Answer Set Programming to reduce the arity of predicates. This is caused by the circumstance that the ASP-paradigm is based on exact matching of a rule's elements and the more parts there exist in a rule, the more effort it is for the ASP-environment to check all of them for equality to the required values.

The second reason for preferring two predicates with smaller arity over one with a higher number of arguments is the fact that it increases the clarity of the code. Although this is hard to prove and depends on the reader's preferences, we are able to focus exactly on the information we need. So, if we do not need any information about the shift's start, we can simply build a rule without the predicate `shift_start`. The same clearly works vice versa too.

Please keep in mind that the approach of splitting a predicate should be tested intensively when implementing own ASP-programs, since the impact on performance can be significant and is strongly dependent on the respective model and also the ASP-environment that is employed to interpret these programs can modify the recommendations.

In our case, we could not find any influence on the overall performance, since there are only very few instantiations of the predicates in our examples used for the evaluations. The decision to stick to the version with two predicates with an arity of four is therefore mainly based on best practice and the personal preference of the author for a clear and well-structured code.

Predicate `required`

```
required(<timeslot>, <requirement>).
```

Predicate 5.2.3: `required`

To complete the knowledge base for our programs, we still need to define the demand of workers for each time slot in the range between 0 and `days * timeslots_per_day`, since otherwise our approaches would be quite pointless. With the predicate `required` we obtain exactly the tool we need for this task.

For every time slot, there must be exactly one instantiation of this predicate, where the first argument `timeslot` is set to the 0-based index of this time slot and the the second argument `requirement` is used to configure the respective demand of workers.

Predicate `optimal_shift_count`

```
optimal_shift_count(<count>).
```

Predicate 5.2.4: `optimal_shift_count`

The pre-calculated optimal count of distinct shifts can be defined via the optional unary predicate `optimal_shift_count`. This information is only used by the program where we search for an exact solution and the optimal count of shifts is already known. The knowledge represented by this fact accelerates the search for a solution significantly, since we only have to satisfy the demand with the given number of shifts. We will have a closer look at the program which relies on this information in the following section.

Resulting representation of the knowledge base

After we have specified all constants and input predicates, we will now have a look at the actual representation of the knowledge base for our introductory example in terms of a listing of the resulting facts.

```
1 #const days = 1.
2 #const timeslots_per_day = 8.
3 shift_start(1, 2, 0, 0).
4 shift_start(2, 4, 0, 1).
5 shift_start(3, 6, 1, 0).
6 shift_length(1, 3, 1, 1).
7 shift_length(2, 3, 1, 1).
8 shift_length(3, 3, 1, 1).
```

Listing 5.2.1: Definition of shift types

```
required(0,1).
required(1,1).
required(2,4).
required(3,3).
required(4,5).
required(5,5).
required(6,2).
required(7,3).

optimal_shift_count(3).
```

Listing 5.2.2: Definition of requirements

5.3 Exact Solution with Known Optimal Shift Count

In this section we will present our modelling approach for instances of the Shift Design Problem where we have full knowledge of the optimal count of distinct shifts and where we search for exact solutions only. As already mentioned, we should keep in mind that this is a very optimistic point of view, since the optimal shift count is in general not known before. Regardless of this limitation, we will use this model as starting point for the explanations of further programs that do not require the additional information of a pre-calculated shift count and thus are much more powerful. In this way, the reader can grow accustomed to the syntax and semantics of the code parts and we can highlight the steps of evolution from the first of our modelling approaches to the flexible, but quite complex, third one. By using this satisfiability problem as starting point for our discussion, we can additionally investigate the impact of optimization statements in terms of performance in chapter 6.

Constant `timeslots`

```
#const timeslots =
    days * timeslots_per_day.
```

Constant 5.3.1: `timeslots`

The code fragment in the box above defines a new constant named `timeslots`, which is a simple alias for the formula `days * timeslots_per_day`. We introduce this new constant since the product of the two input values is needed very often in our programs and it makes the code more readable.

In our introductory example, the formula leads to the calculation $1 * 8 = 8$, so that the number of time slots in the planning horizon is specified with the value 8. Most real-world examples will deal with higher numbers of time slots in all probability. Many of our test instances for the evaluation use a planning period of seven days and at least 24 time slots per day, resulting in a huge search space for our programs.

Predicate `day`

```
day(0 .. days - 1).
```

Predicate 5.3.1: `day`

The predicate `day` is used to enumerate all days in the planning period. One special feature of this rule is the syntax of two points occurring immediately one after another to define an interval.

In this case, the interval starts with 0 and ends with the value `days` - 1, where `days` is the constant already mentioned before. When the rule is resolved by the grounder, we obtain exactly the required n facts, where n is equal to the total number of days in our planning period.

The actual instantiations of the unary predicate `day` always correspond to the 0-based index of the respective day. This means that `day(0)` represents the fact for the first day, `day(1)` stands for the second day and so on. For our introductory example the set of facts obtained after grounding is presented in the following box.

```
day(0).
```

Result 5.3.1: Result of Predicate `day` for introductory example

Predicate `timeslot`

```
timeslot(0 .. timeslots - 1).
```

Predicate 5.3.2: `timeslot`

As already used shortly before, we again employ the convenient syntactical abbreviation for the definition of an interval. At this time, we will utilize it to generate the facts for all possible time slots in the planning period.

Also for this unary predicate we want the index to start with the value 0, which is due to the circumstance that this allows a more flexible usage in the further parts of our programs. We will stick to this choice in all following rules, since it also improves the clarity of mathematical formulae in our upcoming explanations. Result 5.3.2 shows the facts generated by the predicate `timeslot` for our introductory example.

```
timeslot(0).
timeslot(1).
timeslot(2).
timeslot(3).
timeslot(4).
timeslot(5).
timeslot(6).
timeslot(7).
```

Result 5.3.2: Result of Predicate `timeslot` for introductory example

Predicate `change`

```
1 change(0, Requirement1 - Requirement2) :-
2     required(0, Requirement1),
3     required(timeslots - 1, Requirement2).

4 change(Time, Requirement1 - Requirement2) :-
5     required(Time, Requirement1),
6     required(Time - 1, Requirement2).
```

Predicate 5.3.3: `change`

The predicate `change` consists of two rules that are introduced to contribute to an enormous performance improvement compared to a naive approach. The task of these rules is to calculate the changes in demand for each time slot compared to the one right before. This enables us to introduce additional constraints that can be used to restrict the search space for our first two programs.

The first three lines are used to compute the change in demand between the last time slot and the first one, since the planning period is cyclic. In line 2 of Predicate 5.3.3, we check if there is a requirement for the first time slot and store the actual demand in the variable `Requirement1`. In line 3, the same is done for the last time slot, so that we obtain also a value for the second variable `Requirement2`. As already mentioned in Chapter 3 dedicated to the introduction to ASP, the head of a rule evaluates to true, if and only if all literals in the body of the respective rule are satisfied.

Assuming that an instantiation of the predicate `required` exists for the time slot 0 as well as a second one defining the demand at the point in time with index `timeslots - 1`, we have successfully computed a result for this rule. We finally receive a new fact that looks like `change(0, <value>)`, where `value` is the change in demand between the last and the first time slot, which is exactly what we intended with this rule.

The second rule starting with line 4 is the more general one, because the two neighbouring time slots are not fixed any more. Instead, the variable `Time` is used to replace the constant values from the first rule and it represents the index of the time slot currently selected. In line 6, we use the formula `Time - 1` to calculate the index of the earlier time slot. Note that we do not have to take care of negative values, since we have defined the indices of the predicate `required` to be non-negative and therefore the associated literal in the body of this second rule will not evaluate to true. This means that the rule will not fire anyway and we can keep the code nice and simple.

Result 5.3.3 finally shows us the set of facts obtained during the grounding step of our small introductory example. We can see that for every time slot t we have successfully calculated the difference in the amount of required workers between time t and the time slot right before t and this is exactly what we wanted to achieve with this predicate.

```
change(0,-2).
change(1, 0).
change(2, 3).
change(3,-1).
change(4, 2).
change(5, 0).
change(6,-3).
change(7, 1).
```

Result 5.3.3: Result of Predicate `change` for introductory example

Predicate `length`

```
1 length(MinLength .. MaxLength) :-
2     shift_length(_, Length, SlotsAfter, SlotsBefore),
3     MinLength = Length - SlotsBefore,
4     MaxLength = Length + SlotsAfter.
```

Predicate 5.3.4: `length`

In the next step of our modelling approach we define a simple unary predicate that allows us to specify possible shift durations, so that this important information can be used in the body of other rules.

For every instantiation of `shift_length`, this rule generates a set of n facts. Each of these facts is of the form `length(<value>)`, where `value` is a number out of the interval of length n between the minimal and the maximal allowed length. We define the minimum (maximum) duration of a shift as result of decreasing (increasing) the default length of the respective shift by the allowed negative (positive) deviance. The syntactical representation of these two calculation instructions can be viewed in the lines 3 and 4 of the code used to define Predicate 5.3.4. We don't specify an additional argument for the shift type the duration corresponds to, since this value is restricted by upcoming predicates anyway and by omitting additional arguments we can reduce the effort for grounding a little bit.

As we already mentioned, the underscore (_) in line 2 is the syntactical method to inform the ASP-environment that we are not interested in the first argument's value of the instances of the previously defined predicate `shift_length`. By using this syntactical element, we do not need to specify an additional variable. This could lead to slightly less computation effort for the ASP-environment, but the main reason for our choice is that it also makes our code easier to understand.

The actual instantiation of Predicate `length` for our example problem instance is provided in the code listing Result 5.3.4, where we can see for instance that there is no length of zero or only one time slot, since the minimum length of a shift type is defined with two time slots. By removing invalid lengths, the predicate restricts the search space for the further steps of the grounding task.

```
length(2).
length(3).
length(4).
```

Result 5.3.4: Result of Predicate `length` for introductory example

Predicate `amount`

```
amount(0 .. MaxAmount) :-
    MaxAmount = #max [
        required(_, Requirement) = Requirement
    ].
```

Predicate 5.3.5: `amount`

Another very important property of an answer set for the Shift Design Problem is the amount of workers that are assigned to an explicit instance of a shift. The word 'explicit' in the sentence right before is used to highlight that the number of assigned employees is always selected for a fixed starting time slot and also a firm duration, so there is no deviance allowed at all, since this would lead to another version of the shift.

To improve clarity of our explanation we will give a short example. Imagine a shift starting at six o'clock and a default duration of eight hours. We allow at most one hour of overtime. This results in two explicit shifts with both starting at six o'clock, but one of them has a working time of eight hours and the other one has a duration of nine hours. The first instantiation could have an amount of two assigned workers and the second one an amount of six. Although both explicit shifts were generated from the same template, they are treated as completely independent from each other. This means that from 6 am to 2 pm, there are eight employees at work in total.

The highest possible number of workers per explicit shift is defined as the maximum demand in the set of requirements of all available time slots. This is a very relaxed assumption, but it is clearly valid when we want to minimize excess of employees per time slot.

The code excerpt for the rule which we use to enumerate the interval from 0 to the maximum requirement is quite intuitive. We employ the aggregate `#max` to find out the highest number in a set which consists of all values of demand stored in the instances of the predicate `required`. These values are extracted from the mentioned predicate in line 3.

```
amount(0).
amount(1).
...
amount(5).
```

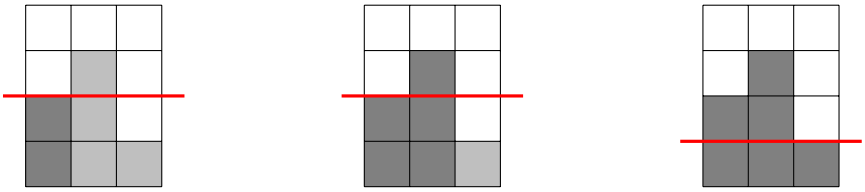
Result 5.3.5: Result of Predicate `amount` for introductory example

Result 5.3.5 shows us again the set of facts obtained in the task of grounding our small example. Since the range of possible values for the number of employees has a significant impact on the program's performance, we will now introduce a further predicate that is used to restrict the actual options for assigning a value for the amount of workers to a explicit shift.

Predicate `min_requirement`

If no excess is allowed for any time slot in the planning period like in our first two models, the easiest way to restrict the number of workers starting at a specific point in time is to calculate the minimum of required employees in a specific lapse of time. For each instantiation of the predicate `shift_start` with start s and length l, this span of time can be defined as interval of time slots that is stretched from time slot s to the point in time with index $s + l - 1$. Please keep in mind that our planning period is cyclic, so we have to take care of cases where the result of the formula $s + l - 1$ exceeds the total number of available time slots.

Again, our model will perform this task without any further effort for the end-user, but before we go into detail, we will give a short example of the intention of our idea to make the following rules more comprehensible.



(a) length: 1 **(b)** length: 2 **(c)** length: 3

Figure 5.3.1: The idea behind the predicate `min_requirement`

Figure 5.3.1 illustrates a small fragment of an sample instance of the Shift Design Problem consisting of three time slots that are depicted by the columns of our grids. For each of these time slots, the grayish boxes represent the current demand at the respective point in time. In our example we act on the assumption that there is exactly one instantiation of `shift_start` with a fixed starting time at 6 o'clock, which is synonymous with time slot index 0. Additionally we presume that the duration of the shift is one hour with an allowed positive deviance of two time slots, where a time slot corresponds to one hour for simplicity reasons.

Every grid illustrates one of the three options for a possible shift duration, where the cells filled with color dark gray correspond to the elements we use for the determination of the value for the minimum requirement, which is represented by the red line.

In the trivial case of the length 1, we can immediately state that the maximum number of employees who are working from 6am until 7am is two, since a higher value would lead to an excess of workers at this time span. This upper bound of workers per explicit shift stays the same for grid (b), since we cannot assign more than two employees to the shift instance from 6 to 8 o'clock without risking excess. For length 3 we are limited to a single worker, since time slot 2 requires only one employee.

When looking at the small example illustrated in Figure 5.3.1, we can see that it really makes sense to restrict the maximum number of employees starting their work at an specific point in time with respect to this additional knowledge of the minimum demand defined for the set of the following time slots.

Another performance improvement is achieved when we only have a look at the relevant instantiations of explicit shifts. We define these relevant instantiations in such a way, that their ending time slot has an requirement greater than zero. The attentive reader could argue that this restriction does not accelerate the task of computation significantly, since the minimum demand in the case of an instantiation of `shift_start`, which is assumed to be irrelevant, will be zero anyway. This calculation is true, since the minimum requirement really is zero in these cases, but there is also an performance improvement, since the effort for grounding can be reduced with this restriction.

The predicate responsible for correctly merging the instantiations of the two input facts `shift_start` and `shift_length` into relevant instances of explicit shifts received the meaningful name `possible_shift_start`. We will give a detailed description of it in the next section of our work, but before we move on to this advanced program element, we will first finish our explanations of the predicate `min_requirement` by having a closer look at the code used to define it.

```
1  min_requirement(Start, Length, Value) :-
2      possible_shift_start(_, Start, Length),
3      EndSlot = Start + Length - 1, EndSlot < timeslots,
4      Value = #min [
5          required(Start .. EndSlot, Requirement)
6              = Requirement
7      ].

8  min_requirement(Start, Length, Value) :-
9      possible_shift_start(_, Start, Length),
10     EndSlot = Start + Length - 1, EndSlot >= timeslots,
11     Value = #min [
12         required(Start .. timeslots - 1, Requirement)
13             = Requirement,
14         required(0 .. EndSlot - timeslots, Requirement)
15             = Requirement
16     ].
```

Predicate 5.3.6: `min_requirement`

We can see that also Predicate 5.3.6 was split into two separate rules, where the first rule is used when the end slot of a shift instance is within the bounds of the planning interval and the second one does the same job for the cases that do not fully fit into this period of time and where the cyclic assignments come into play.

In lines 2 and 9, we use the predicate `possible_shift_start` to find possible instances of shifts. The first argument represents the name of the shift, which is not needed by our rules. The variable `Start` holds the index of the starting time slot, where its value is already expanded to the correct day within the planning period and the variable `Length` is set to a valid shift duration. With this knowledge we can immediately calculate the ending time slot and check if it is within the total number of time slots in the line 3 and accordingly in line 10 for the second rule.

Like in the rule to define the maximum amount of workers, we employ an aggregate to find out the minimum demand. In our rules, the keyword `#min` in the lines 4 and 11 is used to find the minimum requirement in the time span defined by the currently selected instance of a shift. By the way, the comma in line 13 allows us to join multiple sets, so that the aggregate is executed for the set union.

```
1 min_requirement(2,2,3).
2 min_requirement(2,3,3).
3 min_requirement(2,4,3).
4 min_requirement(3,2,3).
5 min_requirement(3,3,3).
6 min_requirement(3,4,2).
7 ...
```

Result 5.3.6: Result of Predicate `min_requirement` for introductory example

Result 5.3.6 summarizes the facts obtained from the predicate while grounding our example of the Shift Design Problem. We can see for instance that for shifts starting at the point in time with index 2 and a duration of four time slots, the minimum number of required workers within this period of time was calculated to be three.

Predicate `possible_shift_start`

As already mentioned before, the predicate `possible_shift_start` merges the knowledge gained from the input facts `shift_start` and `shift_length` into a reusable representation of relevant shift instances, consisting of the name, starting time and duration of the respective shift.

This predicate is a very important part of many rules in our three programs, because it also extends the simple input facts to a more specific form where the sample time slots are put into correlation with the actual day. This means that the starting point in time of a shift is no longer limited by the constant `timeslots_per_day`, but by the value of `timeslots`, since the rules constituting the predicate are evaluated for every single day.

The model for computing the instances of Predicate 5.3.7 is provided in the following code box. For our introductory example the predicate leads to 15 facts in total which will be listed at the end of this subsection. For brevity, we will again only present a subset of the facts.

```
1  possible_shift_start(Shift, Start, Length) :-
2      timeslot(Start), length(Length),
3      shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
4      shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
5      day(Day), Offset = Day * timeslots_per_day,
6      Start >= Start1 + Offset - SlotsBefore1,
7      Start <= Start1 + Offset + SlotsAfter1,
8      Length >= Length1 - SlotsBefore2,
9      Length <= Length + SlotsAfter2,
10     required(Start + Length - 1, Requirement1),
11     Requirement1 > 0.

12 possible_shift_start(Shift, Start, Length) :-
13     timeslot(Start), length(Length),
14     shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
15     shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
16     day(Day), Offset = Day * timeslots_per_day,
17     Start >= Start1 + Offset - SlotsBefore1,
18     Start <= Start1 + Offset + SlotsAfter1,
19     Length >= Length1 - SlotsBefore2,
20     Length <= Length + SlotsAfter2,
21     required(Start + Length - timeslots - 1, Requirement1),
22     Requirement1 > 0.
```

Predicate 5.3.7: `possible_shift_start`

To appease the reader, the rules look much harder to understand than they actually are, since lots of the literals are just used to restrict the range of the the variables `Start` and `Length`, as it can be seen in the lines 6-9 and 17-20.

In each of the rules, the last two lines ensure that the ending time slot has a demand of at least one worker, because otherwise the shift is treated as irrelevant and the respective rule will not fire, causing the instance to be omitted in the task of grounding. This improves the overall performance of our programs in case that points in time with no need for attendant employees exist. To make our explanations complete, we finally want to draw the reader's attention to the lines 5 and 16. These two lines are the originating place where the sample time slots of the two input predicates can be mapped to the indices corresponding to the real points in time within the planning period. We can also see that the choice of a 0-based indexing scheme is quite appropriate for our programs, since we can directly calculate the offset without any additional subtraction.

Following our detailed explanations about Predicate 5.3.7, the reader can find a subset of the facts generated during the grounding step in the code box on the following page. One can easily identify that the rule generates all allowed combinations of shift starts and appropriate durations.

```
possible_shift_start(1,2,2).
possible_shift_start(1,2,3).
possible_shift_start(1,2,4).
possible_shift_start(2,3,2).
possible_shift_start(2,3,3).
possible_shift_start(2,3,4).
...
```

Result 5.3.7: Result of Predicate `possible_shift_start` for introductory example

Predicate `assigned`

We should mention, that all predicates described so far result in facts only. This means that until now everything is determined beforehand and completely resolved during the process of grounding the rules.

To make things more interesting and especially to empower us to actually solve instances of the Shift Design Problem, we will now present maybe the most important rule for the first two implementations where we aim at exact solutions. The task of the following code fragment is to assign a number of employees to each explicit shift. More precisely the rule looks at each time slot t and divides the number of workers required at time t into portions that can be assigned to the shifts that comprise the point in time with index t.

After this short introduction to the ternary predicate `assigned` lets dig a bit into details of the code, since this is also helpful to understand why we had to introduce the lots of other rules mentioned so far:

```
Requirement [
    assigned(Start, Length, Amount) = Amount :
        min_requirement(Start, Length, Requirement1) :
        amount(Amount) : Start <= Time :
        Start + Length - 1 >= Time :
        Amount <= Requirement1,
    assigned(Start, Length, Amount) = Amount :
        min_requirement(Start, Length, Requirement1) :
        amount(Amount) : Start + Length -
        timeslots - 1 >= Time :
        Amount <= Requirement1
] Requirement :-
    required(Time, Requirement).
```

Predicate 5.3.8: `assigned`

There are three syntactical novelties in the code for Predicate 5.3.8. At first we can see that the variable `Requirement` is used twice outside the square brackets. In the input language of our grounder *gringo* the left value is interpreted as lower bound and the right one acts as upper bound for the aggregate that is located between the two integers. The question where the aggregate can be found in the provided rule brings us to the second syntactical refinement we make use of in the code fragment. The language allows us to omit keywords in some cases. One of these special cases are square brackets as well as curly brackets enclosing a set of literals. The square brackets occurring in our rule head are implicitly connoted with the semantics of the aggregate `#sum`. On the other hand, curly brackets without further specification of an aggregate lead to the same behaviour as the keyword `#count` does. The third syntactical novelty is that we move the occurrence of the aggregate we have selected to the head of the rule. By doing so, we employ the very sophisticated mechanisms of the solver *clasp* to generate valid partitions satisfying the provided boundaries.

The superset which is used for this partitioning is the pool of all shifts that enclose the time slot currently selected via the predicate `required`. In our case, we can rely on the minimum requirement as defined before to pre-determine all potential allocations of workers for a shift and we delegate the work of finding a valid selection for the number of workers for each of these shifts to the solver.

Please observe that we do not have to refer to the predicate `possible_shift_start` for this rule since it is guaranteed that for every relevant shift there exists exactly one instantiation of `min_requirement` and so we can obtain the maximum amount directly via the variable `Requirement1` without using any unnecessary additional literals.

Starting from line 7, the reader can see that we have not forgotten the cyclic planning period and also added a subset of values for the cases that the ending time slot is not in the same cycle as the starting slot, so that we really receive the results we expect in every possible case. Before we move on to some additional facts about this important rule, we still should specify the exact mapping of the predicates in the set to the respective numbers acting as terms in the summation. In the lines 2 respectively 7 we can see that this is simply the number of workers assigned to the instance of a shift. When we now recapitulate all the information concerning the rule that defines the predicate `assigned`, we can convince ourselves that this rule enables the solver to partition the number of required workers into portions and assigns these subsets of employees to instances of shifts. This behaviour is perfectly consistent to what we described in the second paragraph of our explanations for this predicate.

Everything we have heard about the rule so far seems to be quite promising, but there is still some explanation needed to fully understand the task of the predicate `assigned`. One very important remark is that we cannot ensure totally valid assignments by now, since every explicit shift is very likely to appear in more than one instantiation of this rule. Assuming that one time slot corresponds to one hour, a shift starting at 6 o'clock and with a duration of eight hours could occur in eight instances of the rule, because the duration encloses eight time slots that need to be evaluated.

One arising problem is that the number of employees assigned to the same explicit shift is not necessarily equal when taking different points in time as origin for this evaluation. It is not hard to avoid such conflicting assignments and we will tackle this issue by further introducing Constraint 5.3.1 below. But before we come to the sections where the constraints are defined, we should mention the other and much harder difficulty of the rule we are currently looking at. It is the enormous size of the search space, since our programs always aim at finding at least one global optimum by testing all possible assignments, depending on the setting of the ASP-environment.

Unlike most other procedural solution approaches the search space is not restricted and so we obtain the advantage that we can always find a global optimum, if there exists one, but on the other hand we have to deal with a huge number of possibilities and therefore time becomes the limiting factor. In chapter 6 we will even see that there are also some instances of the Shift Design Problems that cannot be solved in reasonable time.

Although we have to cope with these two problems, the code provided by us turned out to perform very well, since shifts often interlock in such way that it significantly accelerates the search even in case of a larger search space so that we can kill two birds with one stone. To complete our explanations, we have again listed the instantiations of this rule for our example instance below. For the sake of brevity we only present the resolved rule for covering the time slot with index 0.

```
1#sum[
assigned(7,4,0)=0, assigned(7,4,1)=1,
assigned(6,4,0)=0, assigned(6,4,1)=1,
assigned(7,3,0)=0, assigned(7,3,1)=1,
assigned(6,3,0)=0, assigned(6,3,1)=1,
assigned(7,2,0)=0, assigned(7,2,1)=1
]1.
...
```

Result 5.3.8: Result of Predicate `assigned` for introductory example

Predicate `selected_shift`

```
selected_shift(Start, Length) :-
    possible_shift_start(Shift, Start1, Length), day(Day),
    assigned(Start1, Length, Amount), timeslot(Start),
    Start = Start1 - Day * timeslots_per_day,
    Start < timeslots_per_day, Amount > 0.
```

Predicate 5.3.9: `selected_shift`

After we have guessed the number of workers for each shift with Predicate `assigned`, we will now introduce a new rule to abstract these explicit time spans in such a way that it becomes irrelevant to which day the shift belongs.

This abstracted view is necessary, since in the Shift Design Problem it does not matter on which day a shift is selected. We are only interested in the fact, if it is assigned at least once in the planning period. For instance, if an employee works from 6 am to 2 pm on the first day it has the same effect as if he would work within the same eight hours on any other day in the planning period.

In other words the predicate `selected_shift` boils down the planning period to one day, so that the starting points of the resulting instantiations are within an interval between 0 and the constant `timeslots_per_day`. Explicit shifts with at least one worker assigned are treated as they would also reside on the first day. By doing so, we achieve conformance to the specification of the Shift Design Problem, where a shift counts as selected one if it is used by employees in at least one of the days in the planning horizon.

After these introductory words, let's have a closer look at the code. Starting in the second line of Rule 5.3.9 we use the predicate `possible_shift_start`, so that only relevant shifts are taken into account and the effort for grounding is minimized. Further we rely on the input facts `day` and `timeslot` which are later used to convert the actual starting time of a shift to a time slot within the first day. The explicit shift we are currently looking at is then selected in line 3 via the ternary predicate `assigned` we already explained before quite extensively. By the way, the index of the point in time where the respective shift begins is stored in the variable `Start1`. In the last two lines the actual abstraction is done by mapping the value of `Start1` to an index less than `timeslots_per_day`. In line 5 we additionally ensure that the amount of assigned workers is greater than zero, so that we are really allowed to refer to this abstracted shift as a selected one.

If all literals in the rule's body are satisfied, the grounder creates a new instantiation of the predicate `selected_shift` with the abstracted value of the starting time and the unmodified duration of the original shift. Note that this is the only predicate that is not resolved to a set of facts in the first two programs, because the actual number of workers cannot be determined beforehand. This is a quite remarkable attribute of these two approaches, because with them we can provide two models capable of delegating a lot of the workload to the grounder which leads to a reduced complexity for the solver. Since grounding is a relatively straightforward task, we now have reached the first milestone on the way to an efficient model of the Shift Design Problem. The resulting instances of predicate `selected_shift` can be viewed in the corresponding code box.

```
1 selected_shift(2,2):-assigned(2,2,1).
2 selected_shift(2,2):-assigned(2,2,2).
3 selected_shift(2,2):-assigned(2,2,3).
4 selected_shift(2,3):-assigned(2,3,1).
5 selected_shift(2,3):-assigned(2,3,2).
6 selected_shift(2,3):-assigned(2,3,3).
7 selected_shift(2,4):-assigned(2,4,1).
8 ...
```

Result 5.3.9: Result of Predicate `selected_shift` for introductory example

We can see that this mapping simply reverses the functionality of Predicate `assigned` which is dependent on the respective day of the instance of a shift type. We obtain a new representation that does not use the information about the actual day any more and therefore it can be employed easily to compute the number of selected shifts in the understanding in terms of the Shift Design Problem by our rules that will be presented in the subsequent sections of this chapter. Although this abstraction increases the effort for grounding, during our evaluations it turned out that the overall time needed for solving the benchmark instances actually decreases.

Before we can complete our first modelling approach, we still have to remove invalid answer set candidates and improve the overall performance. We will have a look at this parts of the first program in the following sections that cover the constraints we make use of.

Constraint: Avoid conflicts

The first cause for an invalid answer set we will have a look at is based on conflicting assignments as already described while explaining Predicate `assigned`. Therefore we have to get rid of answer sets containing the same explicit shift multiple times with a different count of assigned workers. This task is common to all our programs presented in this thesis and although it is very important, a solution for this problem can be provided easily. The following code snippet is actually all we need to avoid these sort of conflicts.

```
1 :- assigned(Start, Length, Amount1),
2    assigned(Start, Length, Amount2),
3    Amount1 > Amount2.
```

Constraint 5.3.1: Avoid conflicts

In the box above we can see that the predicate `assigned` occurs twice in the code piece of Constraint 5.3.1. This structure is needed because we have to obtain the ability to check, if there are any ambiguities in the number of workers.

When instantiating the rule, the grounder will generate a constraint for each relevant shift with starting time s and duration d which checks for these conflicts. Since one literal can be matched multiple times by the same instance of a shift, in line 3 we further ensure that the rule only fires in the case that we really have to deal with two different numbers of active workers. In this context, please note that it does not make any difference if we use $>$, $<$ or != as symbol for the arithmetic comparison in line 3 of the code, because the position of a literal in a rule is irrelevant for the result and so we could interchange the two variables `Amount1` and `Amount2` without any influence on the outcome.

Summarizing all the information about Constraint 5.3.1, we can convince ourselves that we have indeed found a constraint which guarantees that the count of employees working for d time slots starting with index s is unambiguous, since it prohibits different assignments of workers for the same time span.

Constraint: Ensure optimal shift count

```
1 :- optimal_shift_count(OptimalCount),
2    not OptimalCount { selected_shift(_, _) } OptimalCount.
```

Constraint 5.3.2: Ensure optimal shift count

After we have successfully avoided conflicts by introducing Constraint 5.3.1, we will now present the constraint that actually empowers us to find solutions for the Shift Design Problem with known optimal shift count.

The basic functionality of Constraint 5.3.2 is to drop all answer sets where the count of selected shifts is not equal to the value of the input fact `optimal_shift_count`. This important value is obtained in line 1 and stored in the variable `OptimalCount`. In the second line we finally define the literal that matches all answer sets where the total count of selected shifts is different from `OptimalCount` so that the rule becomes complete.

Constraint: Ensure correct handling of demand changes

By now our first program is able to do all the things we expect from it and the performance turned out to be quite good on many problem instances, but often this is not enough. Depending on the structure of the problem's actual configuration the time for computing a satisfying answer can become very long. So we tried hard to find some constraint that speeds up the search for a solution significantly without really increasing the effort for the grounder.

Keeping this second requirement in mind is very important, because clearly it does not pay off well, if we save five seconds in computation time when the task of grounding takes the same amount of time longer on the other hand or the amount of main memory needed to store the problem's representation is too high to be handled efficiently. We finally found an approach providing exactly these attributes. Now let's have a look at the details of the implementation:

```
1  :- change(Time, Change),
2     not Change [
3          assigned(Time, _, Amount) = Amount :
4              amount(Amount),
5          assigned(Start, Length, Amount) = -Amount :
6              amount(Amount) :
7              Start + Length == Time,
8          assigned(Start, Length, Amount) = -Amount :
9              amount(Amount) :
10             Start + Length - timeslots == Time
11    ] Change.
```

Constraint 5.3.3: Ensure correct handling of demand changes

The total acceleration effect obtained from employing Constraint 5.3.3 is really impressive, since the constraint restricts the number of possible choices dynamically with every assigned value. Its functionality is actually really easy to explain, because all it does is to ensure that the changes in demand we have already computed via the predicate `change` are also observable when looking the assignments of workers to explicit shifts.

Assuming that the demand at the point in time with index t compared to the one right before is increased by one, then the difference between the number of employees starting their work at time slot t and those who end their work at this time has to be 1 too, since otherwise there would be either a shortage or an excess in the number of assigned workers.

This empowers us to explain why the performance is increased so enormously. Imagine that we need one more worker at time t and further assume that two employees finish their work at this point in time. So we can immediately infer that three workers have to begin their shifts at time slot t. With a growing number of assigned shifts our rule restricts the number of possible choices even more so that the interlocking of these shifts leads to a significant acceleration of the computations.

In line 1 we can see that Constraint 5.3.3 is applied once for every point in time within the planning period, since we already know from before that there is exactly one instantiation of the predicate `change` for every time slot. The second literal is an aggregate computing the total sum over all workers starting or ending their work at the respective point in time.

The number of employees who begin their shift is counted positively for this sum in line 3 and the number of those persons who finish their work is subtracted in the following lines 5 and 8, so that the result of the aggregation represents the actual change in the number of assigned workers for this time slot. Note that the code fragment presented in the lines 8 to 10 takes care of the cyclic planning period while the almost identical code located at the three lines prior to this section is used to handle those shifts where the cyclic structure of the planning interval is not relevant.

The explanations regarding Constraint 5.3.3 given so far can be consolidated by stating that we were able to provide a consistent rule that is matched by all answer sets violating the constraint requiring the actual change concerning the number of assigned workers to be equal to the variation we expect from the problem configuration, so that these undesirable answer sets can be avoided successfully. To the best of our knowledge, the main memory usage is negligible compared to the obtained increase in performance.

Resulting Answer Set for the Example Instance

At this point we have successfully defined all facts, predicates and rules that are necessary for implementing our first modelling approach. Solution 5.3 finally shows the resulting answer set after executing the completed program with the introductory example as input problem instance. Please note that all facts that already have been discussed in the previous sections are excluded in the following listing for the sake of brevity, although they are clearly part of the answer set.

```
1 assigned(2,4,3).   selected_shift(2,4).
2 assigned(4,4,2).   selected_shift(4,4).
3 assigned(7,4,1).   selected_shift(7,4).
```

Solution 5.3: Answer set for introductory example

We can immediately see that the output corresponds to the predicted optimal solution, since we use exactly three shifts with length 4. The indices of the time slots which act as starting points in time are defined by the first argument of the predicates `assigned` and `selected_shift` and the number of employees assigned to each of the shifts according to the third argument of predicate `assigned` are 3, 2 and 1. During our benchmark tests (see chapter 6) we deal with much larger problem instances and the program delivers correct results also in situations with multiple days and a high amount of time slots per planning period.

5.4 Exact Solution with Unknown Optimal Shift Count

In this section we will now focus on the modelling approach that is designed to find answer sets where the amount of required workers is covered perfectly and the count of different shifts is minimal. This allows to be more flexible compared to our first program, since we don't have to know the optimal number of shifts beforehand.

Another advantage of this second approach is that the actual test for minimality is done directly in the context of the solver so that no further external scripts or programs are needed. To clarify this statement, imagine that we want to achieve a similar behaviour by using the first of our approaches. Algorithm 5.4.1 below illustrates one possibility to reach this goal.

Input : An instance of the Shift Design Problem in ASP-syntax I.
Output: The minimum count of shifts needed to solve the problem without any deviance, or the error value -1 in case that I cannot be solved without shortage or excess.

```
1  MaxShiftCount ← 0;

2  foreach ShiftType t ∈ GetShiftTypes(I) do
3  |  MaxShiftCount ← MaxShiftCount + GetNegativeDeviance(t);
4  |  MaxShiftCount ← MaxShiftCount + GetPositiveDeviance(t);
5  |  MaxShiftCount ← MaxShiftCount + 1;
6  end

7  for c ← 0 to MaxShiftCount do
8  |  // Call ASP-environment with first approach and problem instance I:
9  |  // Predicate optimal_shift_count is set to value c.
10 |  if CallASP(I, c) != NULL then
11 |  |  // Problem instance is satisfiable.
12 |  |  return c;
13 |  end
14 end

15 return −1;
```

Algorithm 5.4.1: Sample Algorithm 1

Although the algorithm provided by us looks quite simple, it has at least one really big disadvantage compared to the implementation that is capable to calculate the minimum directly: In the worst case where no zero-deviation solution exists we have to execute the program of our first approach for all possible numbers of different shifts to guarantee that no solution can be found which satisfies our conditions that neither shortage nor excess are allowed. In the algorithm this is represented in line 10, where the function `CallASP` is used to launch the grounder with the respective problem instance and the resulting variable-free set of rules is passed on to the solver by this function. It is assumed that the function performs successful if there exists at least one answer set in the understanding of our first approach for instance I.

Invoking `CallASP` repeatedly causes the problem that the whole grounding task has to be started again for each call of line 10 and also the solver cannot benefit from previously finished reasoning tasks, since it also has to be restarted every time. It should be clear that such ineffective processing is completely undesirable and therefore in practical application scenarios we will always try to find some way so that the expensive grounding task only has to be done once for a problem instance. Therefore we decided to adapt our first modelling approach in such way that it can directly compute the optimal count of shifts that is necessary to solve a given instance of the Shift Design Problem without deviance from the respective requirements. By doing so, we obtain an universally usable tool that can be implemented in almost all ASP-environments, although the syntax might be slightly different.

An additional strength of our second approach is that all grounded instantiations of rules can be kept in memory over the complete processing time of our program, since the newly introduced `#minimize`-statement always stays the same, regardless of the actual optimal number of shifts. In order to achieve the properties we expect, we have to remove Constraint 5.3.2 that is used in our first approach to remove all answer set candidates where the number of selected shifts is different to the specified optimal count and replace it with a proper statement reflecting our need for minimizing those shifts. We use Optimization 5.4.1 to implement this criterion. All other rules remain unchanged and are identical to those in the first approach, therefore we omit them in this section deliberately to avoid redundant explanations.

An alternative approach would be to use some incremental ASP-environment, like *iclingo* [19]. An incremental implementation of an ASP-environment integrates grounder as well as solver into one single program. If the environment is capable of handling incremental programs, the author of such programs can define and use special variables that are automatically incremented if the problem instance cannot be solved with the smaller value. In this way the risk that invariant parts of the input programs have to be grounded repeatedly is minimized as much as possible. For our example problem instance we could exploit incremental ASP by letting the environment choose Predicate `optimal_shift_count` so that Algorithm 5.4.1 is not needed any more. Indeed this would be a pure ASP-solution for the Shift Design Problem with the restriction that only zero-deviation answer sets are allowed. Apart from the fact that in these days there are just a few incremental ASP solvers available, the main problem of reusing the first of our approaches is that still a lot of valuable information could be lost with every incremental step since allowing an additional shift can change the whole set of answer set candidates in the worst case.

Optimization: Minimize shift count

```
1  #minimize { selected_shift(_, _) }.
```

Optimization 5.4.1: Minimize shift count

As mentioned before, Optimization 5.4.1 replaces Constraint 5.3.2 and is responsible for finding the optimal solution for a given problem instance. It is important to keep in mind that an optimization criterion does not prevent the solver from delivering suboptimal answer sets, since the semantics of optimization statements explicitly allow such deviations from the optimum.

We already know that the curly brackets denote that the count of matched instances from the set of literals defined between opening and closing bracket is taken as reference value for the optimization rule.

```
#minimize {
selected_shift(2,2), selected_shift(2,3),
selected_shift(2,4), selected_shift(3,2),
selected_shift(3,3), selected_shift(3,4),
...
}.
```

Result 5.4.1: Result of Optimization 5.4.1 for introductory example

In the box for Result 5.4.1 we can see a fragment of the variable-free representation of the corresponding optimization rule after grounding. One of the solver's most important tasks when processing this rule is to learn new constraints to improve performance for further computation steps on the way to the optimal solution(s) of a given instance of the Shift Design Problem.

It's exactly the process of learning new constraints that makes the second approach more practical compared to the use of the first approach in context of incremental ASP, since for every incremental step, Constraint 5.3.2 will most probably have to be evaluated again and in some cases learned constraints with a positive effect on the overall computation speed might get lost. In contrast, the ASP-environment can learn new constraints more effectively in cases where an optimization statement is used.

Resulting Answer Set for the Example Instance

After we have finished implementing all necessary modifications we can see how easy it actually is to change the semantics of programs in the paradigm of ASP: We changed a few lines of code and the behaviour is totally different, since we do not need to know anything about the number of shifts in our problem instance and we can delegate the whole workload to the solver so that user's comfort increases drastically. Of course we could achieve the same with procedural programming languages, but here the effort will likely be much higher. One can imagine that this advantage of declarative programming will also improve maintainability of the code.

Solutions 5.4.1 and 5.4.2 show two possible answer sets for our introductory example that our second modelling approach could generate. Again we omitted all the facts that are present in every answer set for the sake of brevity.

```
assigned(2,2,3).    selected_shift(2,2).
assigned(4,2,4).    selected_shift(4,2).
assigned(4,4,1).    selected_shift(4,4).
assigned(6,2,1).    selected_shift(6,2).
assigned(7,4,1).    selected_shift(7,4).
```

Solution 5.4.1: One possible answer sets for introductory example

```
1 assigned(2,4,3).   selected_shift(2,4).
2 assigned(4,4,2).   selected_shift(4,4).
3 assigned(7,4,1).   selected_shift(7,4).
```

Solution 5.4.2: Optimal answer set for introductory example

The first answer set depicted in Solution 5.4.1 is obviously not optimal, because five explicit shifts have been selected to cover the demand of workers given by our introductory example. The second answer set shown in Solution 5.4.2 is much better and identical to our well known optimal solution that was also found by our first approach.

Depending on the structure of the problem instance, there can be a lot of steps between the first answer set that is found and an optimal one, but since we do not restrict the search space more than absolutely necessary, we are confident that we always find an optimal solution under the following three assumptions:

- The ASP-environment works as intended
- Sufficient computational resources are available
- A zero-deviation solution exists for the given problem instance

Clearly, the first point in the list above is very likely a base assumption for most programs, because in situations where the underlying infrastructure operates not as expected, almost no software product will execute properly. The second assumption is necessary since the size of a Shift Design Problem is potentially unbounded and with increasing problem size executing the program consumes more and more resources, like processing time and main memory due to the fact that in a pure ASP-environment we always have to take the whole search space into account so that the optimal solution cannot be missed.

Even if our first two assumptions are fulfilled, we still face the problem that in general it is very unlikely that a solution without shortage and excess can be found for the variety of instances of the Shift Design Problem which occur in practice. Therefore we will extend our approach in the next section of our work so that also those instances can be solved where no such perfect solution exists.

5.5 Flexible Solution with Unknown Optimal Shift Count

In the previous parts of this chapter we already presented two ASP-implementations which allow us to find zero-deviation answer sets for instances of the Shift Design Problem where such a solution exists. Since the majority of the problems occurring in practice will most probably require some shortage or excess to be solvable. To give a motivating example for the topic of the general Shift Design Problem, let us have a look at a slightly modified version of our introductory example which was presented some sections before:

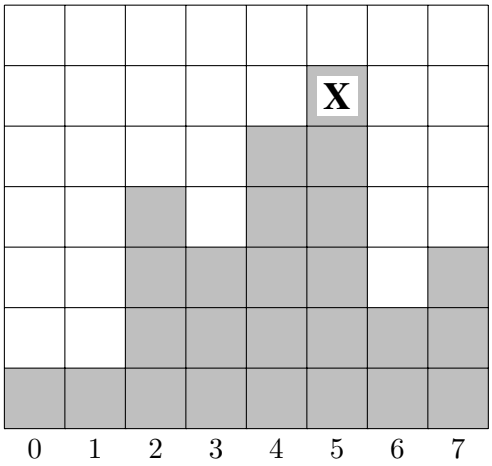


Figure 5.5.1: Initial situation of modified introductory example

The highlighted cell with the letter 'X' inside depicts the facts that an additional employee is required at the time slot with index 5. In comparison, in the original example only five workers are needed at this point in time. This little change in the demand of workers causes our previously presented approaches to fail, since the new problem instance cannot be solved without shortage or excess any more. An informal explanation for this statement can be given in the following way: We know from before that the version with a demand of five employees at the time slot with index 5 can be solved without shortage and excess. Therefore the new version with an additional worker needed at this time slot cannot be solved without deviance from the actual requirements as long as this is the only modification, since each employee has to work for at least two and at most four consecutive time slots. All shifts in our previous sample solution were defined with the maximum length of four time slots so that we have to add at least one additional shift with a working time of two or more time slots. Without proof, we state that adding additional or removing existing instances of the defined shift types also does not help in solving this challenge, since the shape of the distribution of the new requirements in combination with the small set of available shift types does not allow to find a perfect solution for the modified example instance.

With respect to our newly constructed example we realize that our first two approaches could be insufficient for practical usage. To circumvent their limitations, we will present a third approach that theoretically can solve also the general Shift Design Problem where shortage and excess are allowed.

Before we start with the implementation details of our third program, the reader can have a look at the one and only optimal solution for our new problem instance in Figure 5.5.2 which was generated under the assumption that shortage, excess as well as the number of shifts are weighted equally. With a different priority setting it is likely that there are other answer sets considered as optimal ones. In our further explanations we will stick to the usage of equal weights for the key terms shortage, excess and shift count.

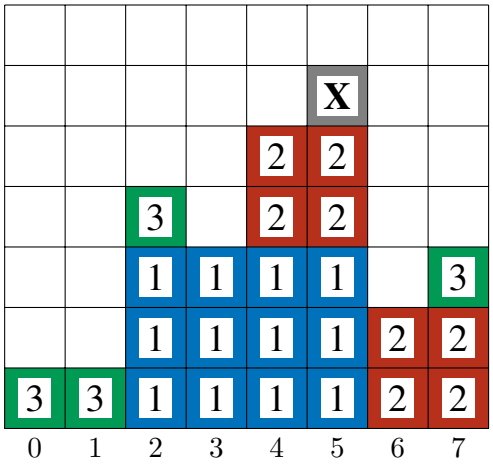


Figure 5.5.2: Solution for the modified introductory example

Initial Situation

In our approaches presented so far, we have introduced a number of constraints so that problem instances where no zero-deviation solution exists are dropped. Since we explicitly allow such imperfect solutions in the new program that will be described in this section, we have to remove those rigorous constraints and change some predicates in order to obtain suitable answer sets for general instances of the Shift Design Problem. To be more precise, only the following rules from the first two modelling approaches will be reused in our third program, while the complete knowledge base of input parameters remains totally unchanged:

- Constant `timeslots` (5.3.1)
- Predicates `day` (5.3.1) and `timeslot` (5.3.2)
- Predicates `length` (5.3.4) and `amount` (5.3.5)
- Predicate `selected_shift` (5.3.9)

Although it seems like a lot code that was used in our previous approaches is useless for the third implementation, that is actually not true, since for most of the predicates that were already discussed in the sections before, minor changes are sufficient to make them well suited for the usage in the depicted program for allowing imperfect solutions.

New Constants for Allowed Deviance

In practice there is a high probability that the management staff wants to avoid some completely undesirable schedules, like time slots where too few or too many employees are at work. To quantify what *too few* respectively *too many* means, we introduce two new constants.

```
1 #const ucover_tolerance = <maximum negative deviance>.
```

Constant 5.5.1: `ucover_tolerance`

```
1 #const ocover_tolerance = <maximum positive deviance>.
```

Constant 5.5.2: `ocover_tolerance`

The value of the constant `ucover_tolerance` defines the maximum negative deviance from the particular demand of workers for each possible time slot and the same holds vice versa for `ocover_tolerance` and the positive deviance.

An example for the impact of Constant 5.5.1 on finding a solution is illustrated in Figure 5.5.3 where `ucover_tolerance` was set to 1 while `ocover_tolerance` is specified with the value 0. In order to keep the size of the search space for our program at a reasonable level, we will use this configuration of the constants for our further explanations.

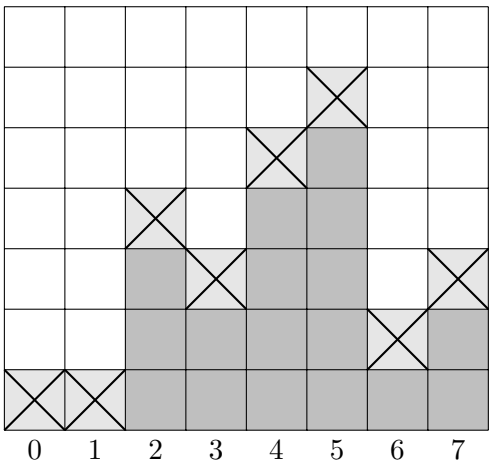


Figure 5.5.3: Relaxed initial situation of modified introductory example

The cells of the grid in Figure 5.5.3 that are marked with a cross represent those parts of the requirements which do not need to be covered necessarily. We can see that the solution for our original example matches this relaxed problem instance almost perfectly. In our case with the given configuration, the solutions for the original introductory example and for the modified one are identical. By comparing Figure 5.1.2 and Figure 5.5.2 we can convince ourselves of this fact.

Predicate `max_requirement`

When we reminisce about the idea of our first two approaches, we observe that a predicate called `min_requirement` has been used to restrict the search space significantly because it definitely makes no sense to send more employees to work than necessary in cases where imperfect solutions should not be taken into account. Since this is desired explicitly in the third implementation, we have to find another way to tame the growing search space.

Predicate 5.5.1 is the tool of our choice for achieving this goal. Obviously it is almost identical to Predicate 5.3.6 `min_requirement` apart from the fact that the keyword `#min` is now replaced with `#max` to calculate the maximum instead of the minimum, therefore we will not go into details too much at this point to avoid redundant explanations. The idea behind this predicate is that it is pointless in the context of the problem statement to have for instance five workers starting at the point in time with index x and a duration of d time slots, if at most three are needed in the time period limited by the points in time with indices x and $x + d - 1$.

Unfortunately, Predicate 5.5.1 limits the search space obviously much worse compared to choosing the minimum as reference value like in our previous programs and so the performance decreases significantly in comparison to our first two modelling approaches as we will see later in chapter 6, where the results of our experimental analysis are presented. Although the impact of this new predicate on the overall performance is quite undesirable, the rule is needed to guarantee that every possible answer set candidate that could lead to a solution can be considered during the processing of the other rules.

```
1  max_requirement(Start, Length, Value) :-
2      possible_shift_start(_, Start, Length),
3      EndSlot = Start + Length - 1, EndSlot < timeslots,
4      Value = #max [
5          required(Start .. EndSlot, Requirement)
6              = Requirement
7      ].

8  max_requirement(Start, Length, Value) :-
9      possible_shift_start(_, Start, Length),
10     EndSlot = Start + Length - 1, EndSlot >= timeslots,
11     Value = #max [
12         required(Start .. timeslots - 1, Requirement)
13             = Requirement,
14         required(0 .. EndSlot - timeslots, Requirement)
15             = Requirement
16     ].
```

Predicate 5.5.1: `max_requirement`

Result 5.5.1 for the predicate with name `max_requirement` shows some instantiations that have been obtained during the grounding process on the left-hand side. The comments on the right-hand side of the same code box were added for explanatory purposes and illustrate the facts that would have been generated by using the predicate `min_requirement` from the previous approaches so that the reader can compare the results of both predicates and get an intuition about the consequential impact on the size of the search space.

```
max_requirement(2,2,4).   % min_requirement(2,2,3).
max_requirement(2,3,5).   % min_requirement(2,3,3).
max_requirement(2,4,6).   % min_requirement(2,4,3).
max_requirement(3,2,5).   % min_requirement(3,2,3).
max_requirement(3,3,6).   % min_requirement(3,3,3).
max_requirement(3,4,6).   % min_requirement(3,4,2).
...
```

Result 5.5.1: Result of Predicate `max_requirement` for introductory example

Predicate `possible_shift_start`

```
possible_shift_start(Shift, Start, Length) :-
    timeslot(Start), length(Length),
    shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
    shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
    day(Day), Offset = Day * timeslots_per_day,
    Start >= Start1 + Offset - SlotsBefore1,
    Start <= Start1 + Offset + SlotsAfter1,
    Length >= Length1 - SlotsBefore2,
    Length <= Length + SlotsAfter2.
```

Predicate 5.5.2: `possible_shift_start`

Predicate 5.5.2 is again pretty much the same as the previously defined Predicate 5.3.7, but in situations where we also want to accept imperfect solutions we are no longer allowed to ignore instances of shift types that have no demand for workers at the last time slot of the respective shift. While we needed two rules to form the old predicate `possible_shift_start`, we only need one rule in the new version of the predicate, since we solely have to provide a mapping for the starting time slots without any need to check the ending time slots that could already reside in the next planning cycle. It is exactly this case distinction that is no longer necessary which allows us to reduce the number of rules composing the new specification of the predicate `possible_shift_start`.

In our new as well as in our older version of the introductory example, there does not occur any time slot where no employee is required to be present, thus the resulting facts generated by Predicate 5.5.2 in the grounding step are the same as for Predicate 5.3.7 for our sample instance:

```
1 possible_shift_start(1,2,2).
2 possible_shift_start(1,2,3).
3 possible_shift_start(1,2,4).
4 possible_shift_start(2,3,2).
5 possible_shift_start(2,3,3).
6 possible_shift_start(2,3,4).
7 ...
```

Result 5.5.2: Result of Predicate `possible_shift_start` for introductory example

Please note that in situations where also points in time exist which have a demand of workers equal to zero, the result will very likely differ and that it is really important for further processing steps to introduce this modified version of the predicate `possible_shift_start` in order to achieve correct results.

Predicate `assigned`

Also in our third modelling approach for the Shift Design Problem, the predicate `assigned` is a central part in the code since it implements the guessing step of the program. To provide correct results on the one hand and to avoid excessive effort for grounding on the other hand, we completely restructured the rule that was used in our first attempts and developed a version that is more suitable for the general Shift Design Problem.

```
1 1 {
2    assigned(Start, Length, Amount) :
3         amount(Amount) : Amount <= MaxAmount
4 } 1 :-
5     max_requirement(Start, Length, MaxAmount).
```

Predicate 5.5.3: `assigned`

We will now explain its basic functionality in more detail. Line 5 of Predicate 5.5.3 illustrates that we solely rely on the previously defined predicate for calculating the maximum requirements in the context of an explicit shift to build the body of the rule, since almost every information we need is stored in the associated instances of the literal. The remaining information is provided by the rule `amount` (see Predicate 5.3.5) so that we are actually able to assign a number of workers to each explicit shift that is available in the input problem instance. In line 2 of Predicate 5.5.3, we restrict the amount of employees per shift in such a way that it has to be less than or equal to the maximum requirement, so that only relevant values are taken into account.

For the head of the new rule we employ the `#count`-aggregate to select one of the elements from the set of possible assignments. We use the value 1 as lower and upper bound in the lines 1 and 4 of the definition of Predicate 5.5.3 in order to achieve this behaviour.

A convenient side-effect of the modified rule is that there is no risk of introducing annoying ambiguities any more. Constraint 5.3.1 is no longer needed and thus, the grounding effort can be reduced. Finally, the listing presented in Result 5.5.3 gives an idea of the resulting rules that are generated for our introductory example in the grounding step of our third modelling approach.

```
1#count{
assigned(2,2,4),
assigned(2,2,3),
assigned(2,2,2),
assigned(2,2,1),
assigned(2,2,0)
}1.
...
```

Result 5.5.3: Result of Predicate `assigned` for introductory example

Predicate `timeslot_cover_value`

```
1  timeslot_cover_value(Time, Value) :-
2      required(Time, Requirement),
3      Value = [
4          assigned(Start, Length, Amount) = Amount :
5              Start <= Time : Start + Length - 1 >= Time,
6          assigned(Start, Length, Amount) = Amount :
7              Time <= Start + Length - timeslots - 1
8      ],
9      Value >= Requirement - ucover_tolerance,
10     Value <= Requirement + ocover_tolerance.
```

Predicate 5.5.4: `timeslot_cover_value`

In order to provide some tool that can be employed to penalize shortage and excess, we make use of Predicate 5.5.4 which calculates the total number of workers that are present at a specified time slot. To minimize the effort for the grounder as much as possible, we do not allow results where the limits introduced by Constant 5.5.1 and Constant 5.5.2 are exceeded.

The actual transformation of this idea into code depicted above represents this restriction in lines 9 and 10. The important variable `Value` which holds the number of workers is calculated in a similar manner than already discussed in the section dedicated to the version of the predicate `assigned` used in the first two approaches (see Predicate 5.3.8), thus we avoid redundant explanations for this part of the rule.

Due to the fact that this rule is evaluated for every time slot in the planning period, we can immediately conclude that those time slots which do not have a corresponding instantiation of the predicate `timeslot_cover_value` violate the restrictions that were defined by using the constants `ucover_tolerance`, respectively `ocover_tolerance`.

Constraint: Remove Answer Set Candidates with Excessive Deviation

To remove all answer set candidates with the undesired attribute of having excessive deviation from the defined requirements, by means of Constraint 5.5.1 we define a new rule that removes all answer set candidates where there exists at least one time slot that has no associated instance of predicate `timeslot_cover_value`.

```
1 :- required(Time, _),
2    0 { timeslot_cover_value(Time, _) } 0.
```

Constraint 5.5.1: Remove answer set candidates with excessive deviance

Line 1 ensures that the rule is evaluated for every time slot where a requirement has been defined. The second line of code presented above matches all those time slots where exactly zero instances of predicate `timeslot_cover_value` can be found within the inspected answer set candidate. With this knowledge we can convince ourselves that the rule actually meets our intention. As we can see, Constraint 5.5.1 is indeed the only constraint that is used in the context of the third modelling approach, since our implementation of the general Shift Design Problem with given maximum deviation limits does not need any further restrictions.

Optimization

After we have successfully removed all answer set candidates which do no correspond to the specified restrictions, we still have to find the optimal answer set(s) to solve the general Shift Design Problem. Obviously, the optimal solution(s) can only be found in situations where some objective measure for the quality of answer sets is defined. With the following three optimization statements we empower the end-user of our third modelling approach to configure this measure flexibly with almost no effort.

Optimization: Minimize Shift Count

In our first optimization criterion we define the fitness value of an answer set with respect to the number of shifts used. In Optimization 5.5.1 this fitness value is defined in the following way: Each instantiation of predicate `selected_shift` is counted as value 1 with the weight 1 and then the overall sum is taken as fitness value. This means, that Optimization 5.5.1 would lead to the result '3@1' for the optimal answer set of our introductory example, since there are three selected shifts.

Of course, the end-user can also choose completely different values for the weight or the formula that is taken as addend. The effects of these changes will be discussed later on in this section.

```
1 #minimize [ selected_shift(_, _) = 1 @ 1 ].
```

Optimization 5.5.1: Minimize shift count

Optimization: Minimize Overall Deviation

The overall goal of the Shift Design Problem also includes that the absolute deviation from the requirements is minimized. In our introductory example we have specified that we use equal weights for all optimization criteria, but this does not necessarily hold for every end-user's needs and so we decided to split up this optimization into two rules: One criterion is used for minimizing the shortage and the other one takes care of time slots where to many workers are present.

In Optimization 5.5.2 we can see a more complex use of the formula that can be used as addend in line 2. By definition, the term *shortage* describes the fact that the number a of available workers at some point in time is smaller the the number r of employees required at this time slot, therefore the difference between a and r is negative. This difference is used as addend for the fitness value and the weight is again set to value 1. Since all addends are negative, we have to use the keyword `#maximize` so that 0 is indeed the optimal value and not the worst. Indeed, we could also have swapped the parts of the formula so that the addends remain positive, but it does not make any difference in terms of performance and so we could present an additional keyword of the input language for the grounder *gringo*.

```
1 #maximize [
2     timeslot_cover_value(Time, Value) =
3         (Value - Requirement) @ 1 :
4         required(Time, Requirement) :
5         Value - Requirement < 0
6 ].
```

Optimization 5.5.2: Minimize shortage

Optimization 5.5.3 works in almost the same way as the rule for minimizing the overall shortage, apart from the fact that here the addends are always positive and therefore the keyword `#minimize` is the right choice for this rule.

```
1 #minimize [
2     timeslot_cover_value(Time, Value) =
3         (Value - Requirement) @ 1 :
4         required(Time, Requirement) :
5         Value - Requirement > 0
6 ].
```

Optimization 5.5.3: Minimize excess

The overall fitness value for an answer set is computed by adding all fitness values, where an arbitrary value of higher weight is always more desirable than every value of smaller weight. Under the assumption that we want to minimize each part of the overall fitness value, an answer set with fitness '5@1, 1@2' is treated as better than one with fitness '1@1, 2@2', since '1@2' is smaller than '2@2'.

Possible Modifications

One possible practical adjustment would be to double the value of the addends for an occurring shortage. In this case a shortage is twice as important for the fitness value as an excess. Another idea to refine the results that are generated by the the third program is to set the weights of the rules for minimizing shortage and excess to a higher value. By doing so, the count of shifts is irrelevant as soon as one answer set approximates the demand better than the current optimal solution.

Resulting Answer Set for the Example Instance

After we have also finished presenting the last modelling approach, in Solution 5.5 we finally provide the optimal answer set for our introductory example that was generated by using our program. As before we omit all predicates that are contained in the every answer set for the sake of brevity.

```
1 assigned(2,4,3).   selected_shift(2,4).
2 assigned(4,4,2).   selected_shift(4,4).
3 assigned(7,4,1).   selected_shift(7,4).
```

Solution 5.5: Optimal answer set for introductory example

The fitness value of this answer set under the given configuration is '4@1', since we have a shortage of one worker at the point in time with index 5 and we use three shifts in total. The mathematical representation of this statement looks like follows:

$$\begin{gathered} ShiftsFitness + ShortageFitness + ExcessFitness \\ = \\ 3@1 + 1@1 + 0@1 \\ = \\ 4@1 \end{gathered}$$

CHAPTER 6

Evaluation of Empirical Results

In this chapter we present the empirical results obtained by our modelling approaches. We will first introduce the different sets of problem instances we use for our experiments and afterwards we will give an overview of the software and hardware environment we employed during the evaluation. In the subsequent section we are going to present the results of our experiments and provide a comparison between the solvers *clasp* and *unclasp* from the Potsdam Answer Set Solving Collection. Afterwards we will have a look at previous results obtained by Di Gaspero et al. in [13]. Finally, we will conclude this chapter by recapitulating the outcomes of our experiments.

6.1 Problem Instances

The benchmark we used for this work consists of four different sets of problem instances for the Shift Design Problem. Subsequently, we will briefly explain the basic structure and the most important characteristics of each of these four sets. For interested readers and for potential comparative analyses, all benchmark sets are publicly available under the address `http://www.dbai.tuwien.ac.at/proj/Rota/benchmarks.html`. The data sets were first described in [34, 35]. Di Gaspero et al. [14] used the same instances for the evaluation of hybrid approaches for the Shift Design Problem. During our research, all problem instances were converted to an ASP-compatible format in the input language of *gringo* [21] and they are available under the following address:

`http://www.dbai.tuwien.ac.at/proj/Rota/DataSetASP[1-4].zip`

DataSet1

The first data set contains 200 instances of the Shift Design Problem. The problem instances of DataSet1 can be solved without any deviation, since they were generated by first constructing a feasible assignment of workers to a selected number of shifts (also called the *seed solution*) and then the resulting values were used as requirements for the respective instance.

DataSet2

The second data set contains 30 instances which are quite similar to those of the first data set, but here the seed solution was constructed in such a way, that instances 1 to 10 should need at least 12 shifts to be solved exactly. The instances 11 to 20 feature 16 shifts and the remaining ten instances were constructed with a seed solution that uses 20 shifts. Di Gaspero et al. [13] note, that their heuristic could also find better results for solving the problem instance in some cases. For our computational evaluation, we will therefore use their results as starting point for the comparison. This second data set was originally constructed with the intention to study the relation between the number of shifts in the best known solutions and the computation time of the programs.

DataSet3

Di Gaspero et al. [13] highlight that in cases where an exact solution exists, the behaviour of heuristics could be biased in comparison to the general case where no solution without deviation exists. To allow observations about the behaviour of solvers for the Shift Design Problem in the presence of instances which cannot be solved exactly, the third data set was constructed. It contains 30 instances that were constructed in the same way as the two previous data sets, but at this time, invalid shifts were added during the construction process. These invalid shifts cannot be selected during the computation of the optimal solution, so that it is unlikely that an instance of the third data set can be solved without deviation. The instances 1 to 10 were constructed with a seed of 12 shifts (valid and invalid ones) and also the remaining instances were generated using the same scheme concerning the number of shifts as in the second data set.

DataSet4

The fourth set contains three advanced problem instances, where the first one is a real-world example that can be used to compare the performance of the programs between finding the optimal solution for real-world problems and solving randomly generated instances. The second instance is almost identical to the fifth one in DataSet3, but the length of a timeslot is halfed. In this way, the second instance can be used to investigate the impact of increasing the granularity of the segmentation of the planning horizon. The same holds for the third instance, but here the requirements are doubled instead of the number of time slots.

6.2 Experimental Setting

The experiments in this work were designed to evaluate the following performance parameters of our modelling approaches for the Shift Design Problem presented in the previous chapter:

1. Time needed to reach the best known solution

2. Objective value obtained within the execution time

For computing the objective value of the second parameter we use the same formulation as starting point as it was used by Di Gaspero in [13] to ensure that our results are comparable. The formula used by Di Gaspero et al. to calculate the fitness value of a given solution consists of three components, namely F_1 (excess), F_2 (shortage) and F_3 (number of shifts). The first two components are computed by multiplying the total number of workers in excess/shortage with the length of a single time slot in minutes. The value of F_3 is multiplied with the length of a time slot in minutes, so that the penalty of each shift is equivalent to one worker in excess/shortage for one time slot.

One goal of this work is to investigate, how suitable the programming paradigm of ASP is for solving the Shift Design Problem. Therefore we will compare the solutions generated by our modelling approaches for DataSet1 and DataSet3 with the results that were obtained in [13].

The following two tables describe the most important parts of the system configuration that was used to run our experiments. Although the processor of our testing system was capable of multi-threading, we used the single-threaded versions of the grounder and the solvers in order to simplify comparative evaluations, since most of the state-of-the-art ASP-environments use only one thread at a time.

Processor:	Intel Xeon E5345 @ 2.33GHz
CPU-Cores:	8
Main Memory:	48 GB

Table 6.2.1: Hardware Configuration

Operating System:	opensuse 11.4 (64bit)
Grounder:	*gringo 3.0.4* (single-threaded)
Solver 1:	*clasp 2.1.0* (single-threaded)
Solver 2:	*unclasp 0.1.0* (single-threaded)

Table 6.2.2: Software Configuration

In our experiments, the grounder *gringo* [21] as well as the solvers *clasp* [22] and *unclasp* [1] were executed without changing any of the program parameters. The only exception is the solver *unclasp* in combination with our first modelling approach. Since the first one of our three programs does not use any optimization statements, to avoid problems we have to use the parameter `opt-uncore=no` due to the fact that *unclasp* expects at least one optimization criterion by default.

In the following sections of our work, we will use the names “Exact1” and “Exact2” and “Flexible” for our three modelling approaches. Since an inherent expectation of the first two of them is that a zero-deviation solution for a given problem instance exists, we use only the first two data sets for the experimental evaluation of these modelling approaches.

6.3 Computational Results for DataSet1 and DataSet2

In this section we will present the results that were obtained during our benchmarks. The goal of the first experiment was to investigate the performance of our implementations in terms of the time needed to compute the best known solution for the problem instances of DataSet1 and DataSet2. Due to the huge number of tests, we only run one trial per instance and program, therefore the obtained results should only be takes as indicatory.

In Table 6.3.1 we present the overall time (including the time needed for the grounding task) needed by our programs to compute the best known solution (given in the second column) for the first 30 problem instances of DataSet1. The columns of the tables represent the combination of the modelling approach and the solver used to obtain the respective results. The dashes which can be found in some of the cells are used to highlight those experiments, where the best known solution could not be found within a given time limit. For the two programs where no deviation from the workforce requirements is allowed, namely *Exact1* and *Exact2*, we granted an execution time of one hour. For the modelling approach *Flexible* with a certain amount of deviation allowed, we have limited the permitted execution time for each of the experiments to 30 minutes. The decision to use a more restrictive time limit for the program *Flexible* is caused by the assumption that in practical cases, different combinations of maximum positive and negative deviation will be tested and therefore the time limit for each of the tests will be likely smaller than the limit that is used for the those approaches, where only one test run is needed to cover all possibilities. We reuse this configuration for all of our experiments.

The benchmark values in the last two columns of Table 6.3.1 were obtained by using 0 as maximum allowed positive/negative deviation in the number of workers per time slot. This means that we still only allow exact solutions, since this can be assumed to be the general starting point for using the third program.

Immediately we can see, that the performance in terms of computation time of the flexible approach is much worse compared to the first two programs in most cases. We assume that this is caused by the lack of constraints, since our three implementations are quite similar when in all of them only exact solutions are allowed. The main difference between our second modelling approach and the third one is the fact, that the latter uses three optimization criteria instead of only one in the first case and the number of integrity constraints within the programs is reduced. These circumstances lead to the handicap, that the efficiency of investigating the search space decreases. Additionally, the two optimization criteria, which are used to minimize shortage and excess in the third approach, significantly increase the complexity.

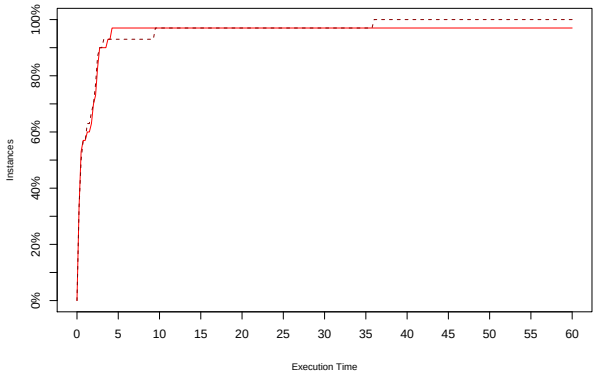
This point of view with respect to the differences of the three approaches allows us to give an explanation, why the third program is sometimes remarkably faster in solving some of the problem instances, like 6, 15 and 30. In these cases, additional experiments on the first two data sets, which are omitted for brevity, showed that grounding is the harder part for these instances while solving can be done quite fast. The accelerating effect of the additional rules, like Constraint 5.3.1 and Constraint 5.3.3, cannot be exploited in these situations, since the effort for grounding the above mentioned rules is higher than the amount of time saved due to their characteristics.

Instance	Best	Exact1		Exact2		Flexible	
		clasp	unclasp	clasp	unclasp	clasp	unclasp
1	480	2.71	2.84	2.72	2.83	3.75	3.61
2	300	22.21	23.48	22.50	22.48	—	988.01
3	600	2.54	2.61	2.57	2.61	141.62	180.78
4	450	35.78	35.78	86.97	26.75	—	—
5	480	3.15	3.17	3.15	3.17	9.57	2.66
6	420	2.44	2.53	2.44	2.55	1.93	0.89
7	270	19.79	20.27	19.77	20.57	—	—
8	150	130.15	159.56	154.01	158.99	—	—
9	150	152.32	130.62	129.74	132.36	—	—
10	330	23.35	23.80	23.88	23.19	—	—
11	30	110.86	113.20	111.02	114.42	14.01	14.20
12	90	99.21	101.39	98.53	100.65	—	183.94
13	105	135.24	139.54	135.55	139.55	—	—
14	195	212.04	2149.95	2320.62	—	—	—
15	180	2.39	2.40	2.40	2.40	0.21	0.23
16	225	241.37	568.48	251.17	169.23	—	—
17	540	—	73.77	—	27.79	—	—
18	720	3.61	3.66	3.61	3.84	—	—
19	180	117.29	121.20	118.72	126.12	—	—
20	540	3.25	3.45	3.25	3.36	4.00	1.36
21	120	146.64	146.93	144.82	151.48	—	—
22	75	72.31	74.02	72.10	73.39	101.05	161.72
23	150	163.12	181.13	299.63	167.85	—	—
24	480	3.20	3.27	3.21	3.36	6.87	3.34
25	480	27.51	40.11	52.11	26.38	—	—
26	600	3.55	3.58	3.71	3.93	—	698.61
27	480	3.37	3.65	3.36	3.66	73.50	193.06
28	270	20.78	21.07	20.78	21.39	—	—
29	360	22.20	22.65	22.55	23.10	—	—
30	75	141.78	146.20	140.80	143.89	23.23	14.05

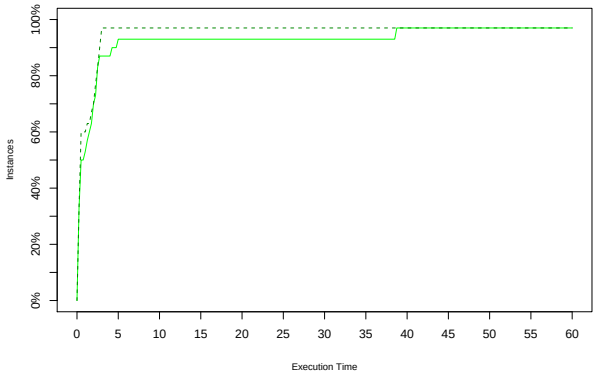
Table 6.3.1: Times (in seconds) to reach the best known solution for DataSet1

Our experiments show that especially for instances where a zero-deviation solution exists, the results obtained using ASP are quite encouraging, since our specialized programs *Exact1* and *Exact2* almost always find the optimal solution for the given problem instances within the allowed execution time of one hour.

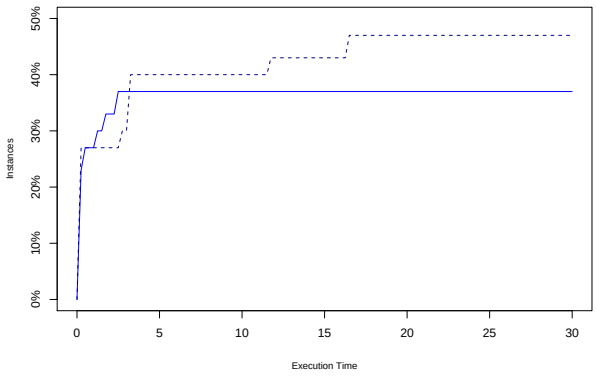
Subsequently, Figure 6.3.1 depicts the percentage of the first 30 instances from the first data set where the best known solution could be reached in relation to the allowed execution time of the programs. In all following figures, the dashed line illustrates the results obtained by the solver *unclasp* and the continuous line is used to present the results for *clasp*.



(a) Exact1



(b) Exact2

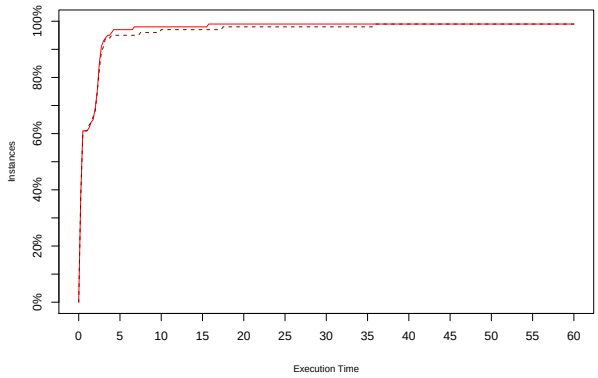


(c) Flexible

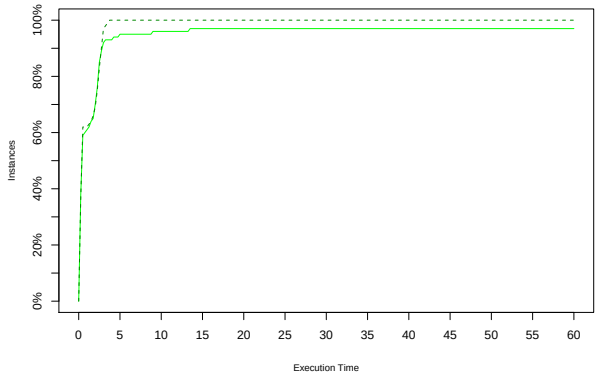
Figure 6.3.1: Solved instances per execution time (in minutes) for first 30 instances of DataSet1

Results for *clasp* are illustrated by the continuous line and the dashed line represents *unclasp*.

In Figure 6.3.1 we can observe that *clasp* performs slightly better in the first 35 minutes for *Exact1* on the one hand. On the other hand, *unclasp* is able to find all of the best known solutions within the time limit of one hour, while *clasp* cannot find the optimal solution for one problem instance. For our program *Exact2*, *unclasp* is superior, although both solvers cannot find the best known solution for one of the instances. For the flexible modelling approach, *unclasp* seems to provide better results at the first glance, but both solvers have problems to tame the enormous search space, so that more than 50% of the instances cannot be solved within the time limit of 30 minutes.



(a) Exact1



(b) Exact2

Figure 6.3.2: Solved instances per execution time (in minutes) for complete DataSet1

Results for *clasp* are illustrated by the continuous line and the dashed line represents *unclasp*.

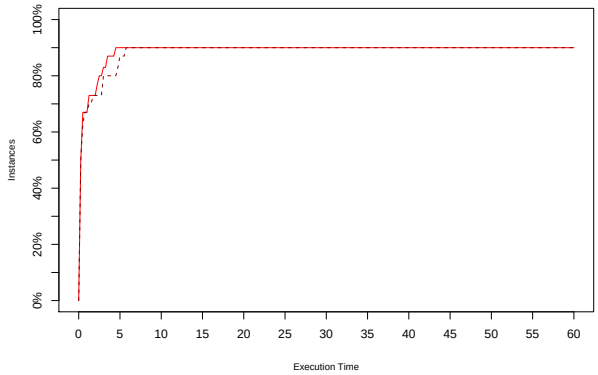
To confirm our statement about the encouraging characteristics of the first two modelling approaches, Figure 6.3.2 finally depicts the experimental outcomes in terms of the percentage of solved instances for the complete first data set in combination with the two aforementioned programs.

DataSet1 consists of 200 problem instances which can be solved without deviation and by using the whole set, we expect a more representative analysis of the logic programs *Exact1* and *Exact2*. At this time, we decided to omit the benchmarks for the third approach, since we know in advance that further improvements for its implementation are necessary to allow a fair comparison between the two solvers.

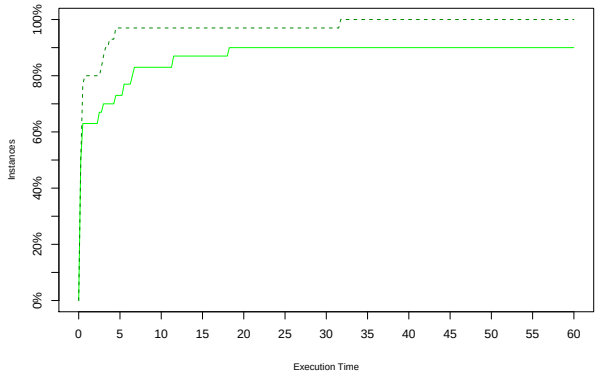
Instance	Best	Exact1		Exact2		Flexible	
		clasp	unclasp	clasp	unclasp	clasp	unclasp
1	720	3.46	3.70	3.52	3.57	871.53	202.67
2	720	3.56	3.69	3.64	3.61	—	—
3	360	23.81	24.07	26.39	23.46	—	—
4	360	22.11	22.54	22.27	22.42	—	1790.58
5	720	3.53	3.67	3.67	3.58	—	1285.70
6	360	22.57	23.65	22.51	22.78	—	—
7	720	3.65	3.91	3.70	3.72	—	—
8	180	165.42	165.44	263.03	156.28	—	—
9	360	22.60	23.31	22.66	22.51	—	—
10	660	3.84	4.18	3.97	3.86	—	—
11	480	126.16	274.68	689.62	25.09	—	—
12	900	3.96	4.16	4.90	4.16	—	—
13	900	4.36	4.68	4.99	5.71	—	—
14	840	3.68	3.92	4.05	3.91	—	—
15	480	27.99	299.31	399.88	26.24	—	—
16	240	195.10	179.00	176.23	169.93	—	—
17	960	3.70	3.89	3.85	3.78	—	—
18	840	3.87	4.14	4.81	4.72	—	—
19	240	257.59	332.71	1092.72	213.50	—	—
20	960	3.85	4.02	4.00	3.89	—	—
21	600	70.72	96.69	142.89	28.03	—	—
22	1080	4.06	4.61	5.16	5.18	—	—
23	300	—	—	—	259.45	—	—
24	600	136.38	73.15	324.96	33.13	—	—
25	600	65.10	37.74	384.02	29.47	—	—
26	1020	4.02	4.01	4.79	4.33	—	—
27	300	—	—	—	1895.29	—	—
28	300	—	—	—	181.34	—	—
29	1140	3.91	4.45	4.77	4.23	—	—
30	1020	5.19	13.95	7.11	5.56	—	—

Table 6.3.2: Times (in seconds) to reach the best known solution for DataSet2

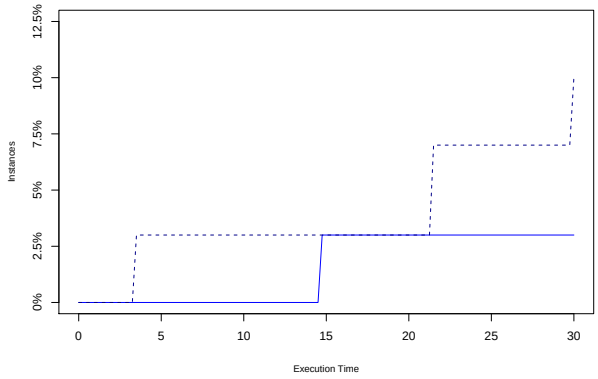
Table 6.3.2 illustrates the solution times for the second set of instances. Again, we can see that the first two programs are significantly faster for instances which can be solved without deviation in the majority of the cases, which conforms to the observations from the experiments of the first data set.



(a) Exact1



(b) Exact2



(c) Flexible

Figure 6.3.3: Solved instances per execution time (in minutes) for complete DataSet2

Results for *clasp* are illustrated by the continuous line and the dashed line represents *unclasp*.

For the second data set, illustrated in Figure 6.3.3, the difference in terms of the percentage of solved instances within a given time bound between the solvers *clasp* and *unclasp* is a bit more distinctive. We can see that the increased complexity in terms of a higher count of shifts is a source of hardness for *clasp*, while *unclasp* can again solve all instances with the program *Exact2*. For the first of our ASP implementations, the performance of the two solvers in terms of computation time is again similar, since we had to disable the promising uncore-algorithm, as mentioned before. Unfortunately, *clasp* fails to compute the best known solution for 29 out of 30 instances with our program *Flexible* and *unclasp* is only slightly better with three solved instances in total.

An additional interesting finding of our experiments is the fact, that the solver *unclasp* is often faster and finds more solutions within the time limits than *clasp* for the second one of our modelling approaches. However, when having a look at the program *Exact1*, the solver *clasp* appears to be performing a bit better. We will have a look at this behaviour also in the summary at the end of this chapter.

Unfortunately, the third program cannot find the best known solution within the time limits for most of the instances of DataSet2. We assume that the main problem of the program *Flexible* is that due to the lack of implemented constraints, the heuristics within the solvers do not have enough information to efficiently control the search procedure by learning new constraints and exploiting the implied knowledge base.

6.4 Computational Results for DataSet3 and DataSet4

In this section we will analyze some instances of DataSet3 and DataSet4 where the time bound for the solvers was set to 30 minutes and we present the best fitness value that could be obtained by our program *Flexible* on these problem instances with an maximum negative deviation of seven workers per time slot. Furthermore, we investigated the differences of the two solvers *clasp* and *unclasp* with respect to the outcome of our experimental evaluation.

Excess	DataSet3 Instance 4	DataSet3 Instance 14	DataSet3 Instance 24	DataSet4 Instance 2
0	—	—	—	—
1	—	—	—	38010
2	27330	—	—	**32130**
3	**25200**	**30000**	35400	32880
4	30630	31740	**33180**	32610
5	29100	33180	36300	34830
6	38370	33780	35760	33450
7	38760	35520	37080	37170

Table 6.4.1: Objective value obtained within 30 minutes for selected instances

Table 6.4.1 shows the fitness values of the solutions for the instances 4, 14 and 24 from the third data set and of instance 2 from the fourth data set. They were obtained by using the third modelling approach in combination with the solver *clasp* and the maximum allowed shortage per time slot was set to 7. The best fitness value that could be found by our programs for each of the instances is highlighted in boldface. Unfortunately, *unclasp* could not find any answer sets within 30 minutes for any instance within DataSet3 and DataSet4 with the program *Flexible*.

We have chosen these four instances mentioned above as they provide a good starting point for further investigations, since one can see that in some cases, the best fitness value found by *clasp* is achieved with a slightly relaxed limit of allowed excess. During our further research, analyses of the generated output of the solvers have shown that this is caused by the fact, that the optimization process clearly cannot be started until the first answer set is found.

This also explains why *unclasp* cannot find any solution within 30 minutes, since the built-in heuristics of this solver start with a very restrictive initial value for the internal bounds of the optimization criteria (e.g. 0 for `#minimize`-statements) and then try to adapt these bounds until the first answer set is found, while *clasp* starts from a relatively high initial objective value and afterwards it tempts to decrease this value until the optimum is reached.

For the instances of the Shift Design Problem in DataSet3 and DataSet4, *clasp* produces more answer sets. Especially with relaxed limits, the solver can compute the first answer set faster and this explains, why sometimes the better results are obtained with these slightly relaxed values for the maximum allowed deviation. The restrictive initial bounds of *unclasp* in general provide more promising characteristics for the first two data sets, as we could see in the section before.

In the following sections of this chapter we will have a look at the overall performance characteristics of our programs by using existing results as starting point for a comparison in order to find out, how competitive our implementations are.

6.5 Comparison with Previous Results

After investigating the results obtained by our programs in detail, in this section we will compare them with experiments presented previously in the literature. To be more exact, we use results provided by Di Gaspero et al. [13] for the comparative evaluation. Di Gaspero et al. investigated the performance of their solvers for the Shift Design Problem with the first and the third data set which were also used in our work.

We note that the differences in the execution times of our programs implemented using ASP and their heuristics should be treated with caution, since a different system configuration was used for the experiments in [13]. Nevertheless, a comparison allows us to study the strengths and weaknesses of our approaches in terms of the solution quality.

In [13, 35], the following implementations were investigated:

- **LS (Local Search):**

  A local-search procedure with multiple neighbourhood relations.

- **GrMCMF (Greedy MCMF):**

  A greedy heuristic using a polynomial min-COST max-FLOW subroutine.

- **GrMCMF+LS (Greedy MCMF + Local Search):**

  A hybrid approach, where the solutions generated by GrMCMF are used as starting points for local search. By starting the local search procedure with an adequate initial solution, the risk to get stuck in neighbourhoods with a solution quality far away from the optimum decreases significantly. During their research, Di Gaspero et al. [13] identified this approach to be superior to LS and GrMCMF as standalone applications.

For our comparative evaluation, we consider the results described in [13] and provide an overview of two experiments. At first, we compare the time needed to reach the best known solution for instances of the first data set. In the second experiment, we will have a look at the third data set in order to check the quality of the generated results when no solution without deviation from the actual staff demand exists. In the first case, we use the program *Exact2* for comparison with the existing solvers, since it performed best during our experiments. In the second case, we have to use the program *Flexible* as this is the only program developed by us which can deal with non-exact solutions.

Table 6.5.1 illustrates the results that were obtained by Di Gaspero et al. [13] by applying their heuristics to the first 30 problem instances of DataSet1 in comparison with our second modelling approach. The test results taken from the previous experiments in [13] are average execution times over ten trials, while we used only one attempt. We stick to one trial since during development it turned out that the standard deviation of the time needed to find the optimal solution is negligible when the default parameters of the solvers are used. Again, the dash in some cells of the table denotes the fact that the optimal solution could not be found in any of the trials.

Apparently, the performance of the current versions of our modelling approaches for the Shift Design Problem is often worse when compared to the results that have been obtained in [13] using heuristic-based approaches. At this point, we should remind that the programs in the paradigm of ASP still are primarily a tool for exhaustive search. The fact that these programs have to investigate the whole search space clearly leads to a deterioration in terms of performance, but this understanding should not make us forget the fact that for practical cases the method is not robust enough at the moment. Especially when the number of time slots raises above a certain level, the solution time is no longer competitive.

Instance	Best	GrMCMF [13]	LS [13]	GrMCMF+LS [13]	Exact2	
					clasp	unclasp
1	480	0.07	5.87	1.06	2.72	2.83
2	300	—	16.41	40.22	22.50	22.48
3	600	0.11	8.96	1.64	2.57	2.61
4	450	—	305.37	108.29	86.97	26.75
5	480	0.20	5.03	1.75	3.15	3.17
6	420	0.06	2.62	0.62	2.44	2.55
7	270	1.13	10.25	6.95	19.77	20.57
8	150	—	18.98	10.64	154.01	158.99
9	150	3.53	11.85	8.85	129.74	132.36
10	330	—	66.05	84.11	23.88	23.19
11	30	0.21	1.79	0.85	111.02	114.42
12	90	0.25	6.10	3.84	98.53	100.65
13	105	0.35	7.20	3.82	135.55	139.55
14	195	—	561.99	60.97	2320.62	—
15	180	0.04	0.89	0.40	2.40	2.40
16	225	—	198.50	151.78	251.17	169.23
17	540	—	380.72	288.42	—	27.79
18	720	1.71	7.72	7.32	3.61	3.84
19	180	—	38.33	31.12	118.72	126.12
20	540	0.11	15.24	1.69	3.25	3.36
21	120	0.28	6.19	2.18	144.82	151.48
22	75	0.65	3.67	3.80	72.10	73.39
23	150	6.19	19.16	22.15	299.63	167.85
24	480	0.11	2.85	1.44	3.21	3.36
25	480	—	503.40	—	52.11	26.38
26	600	1.50	9.59	9.20	3.71	3.93
27	480	0.07	4.02	2.34	3.36	3.66
28	270	2.24	9.25	3.81	20.78	21.39
29	360	—	20.59	10.00	22.55	23.10
30	75	0.26	2.78	1.95	140.80	143.89

Table 6.5.1: Times (in seconds) to reach the best known solution for 30 instances of DataSet1

Another important outcome of the first experiment is, that although the execution times of our second approach are often not competitive with previous literature, some instances (e.g. 4, 10 and especially 25) can be solved with ASP very well. Therefore, we assume that ASP-based implementations for the Shift Design Problem have a high potential to be investigated in more detail in future work.

Instance	GrMCMF [13]	LS [13]	GrMCMF+LS [13]	Flexible	
				clasp	unclasp
1	2445.00	9916.35	2386.80	—	—
2	7672.59	9582.00	7691.40	30390	—
3	9582.14	12367.50	9597.00	26490	—
4	6634.40	8956.50	6681.60	25200	—
5	10053.75	10311.60	9996.00	23100	—
6	2082.17	4712.25	2076.75	10740	—
7	6075.00	12251.70	6087.00	—	—
8	9023.46	10512.60	8860.50	27930	—
9	6039.18	11640.60	6036.90	—	—
10	2968.95	4067.10	3002.40	13770	—
11	5511.43	7888.20	5490.90	34710	—
12	4231.96	11410.05	4171.20	—	—
13	4669.50	10427.55	4662.00	—	—
14	9616.55	10130.40	9660.60	25680	—
15	11448.90	13563.60	11445.00	34980	—
16	10785.00	11180.40	10734.00	22020	—
17	4746.56	11735.40	4729.05	—	—
18	6769.41	9516.60	6692.40	23700	—
19	5183.16	10825.20	5157.45	—	—
20	9153.90	12481.80	9174.90	33720	—
21	6072.86	14102.55	6053.55	—	—
22	12932.31	16418.70	12870.30	39300	—
23	8384.24	9788.40	8390.40	24840	—
24	10545.00	11413.20	10417.80	29520	—
25	13204.80	14038.80	13252.20	33540	—
26	13152.73	17326.50	13117.80	—	—
27	10084.94	10866.60	10081.20	—	—
28	10641.21	11543.40	10603.80	26760	—
29	6799.41	12075.30	6690.00	23910	—
30	13770.68	14808.60	13723.80	32700	—

Table 6.5.2: Comparison of solution costs for the complete DataSet3

In Table 6.5.2, we present the best fitness value that could be obtained by using our modelling approach *Flexible*. Although a direct comparison is not representative as in [13] only one second of execution time was permitted while we used a time limit of 30 minutes, the table shows that our third approach is not yet giving good results for practical instances of the Shift Design Problem.

The data in Table 6.5.2 which was taken from [13] are average fitness values over 100 trials. Our results, which are shown in the last two columns of the table, represent the minimum fitness value over 64 test runs which were organized in such a way that the program *Flexible* was called once with every possible combination out of `ocover_limit` and `ucover_limit`, both with an integer domain from 0 to 7. Each of our trials was granted a maximum execution time of 30 minutes.

Unfortunately, *unclasp* could not compute any answer sets for the given problem instances, since the first answer set found is assumed to be the global optimum and it is unlikely that it can be computed within 30 minutes of execution time. The results generated using *clasp* which are presented in the table are the best fitness values obtained over all 64 trials per instances.

6.6 Summary of the Experiments

As in almost any declarative language, the process of interpreting the obtained results is hardened significantly, since the solvers are a kind of "black box". This means that we were limited to interpreting the debug output (obtained with the parameter `verbose`) in order to find out, what happens inside the solvers on the way to the optimal solution. In cooperation with the University of Potsdam, the place of development of the solvers used in this work, we concluded that the main source of hardness for our programs are the nested aggregation rules in combination with the optimization statements. The term "nested aggregation" is used by us to describe the fact that for determining the fitness value, at first the difference between the number of assigned workers and the requirements has to be calculated for each time slot and afterwards, the sum of the absolute values of these differences has to be computed. Furthermore, at the moment the solvers cannot infer that, for instance, selecting three workers for a time slot x automatically implies selecting two workers for time slot x. This can prevent the internal heuristics from working at highest efficiency. That the investigation of the performance of solvers in relation to the provided instances is a hard task is also admitted by Di Gaspero et al.: *In conclusion, a characterization of the instances in terms of their features is not a simple issue for MSD*[1] *and it needs additional analysis on the instances in order to obtain a precise picture.* [13, p. 27]

In our experimental evaluation, *unclasp* shows increased robustness against changes in the structure of the problem instances for the first two modelling approaches, while *clasp* is superior in most cases when the flexible program with allowed deviation is used (and indeed, the allowed positive/negative deviation is greater than zero). According to the intuition we have obtained during our experiments, especially this third approach needs additional attention in potential further refinement iterations in order to improve its performance and robustness attributes.

In conclusion, *unclasp* seems to provide best performance when optimization is needed and the corresponding criteria are based on simple predicates. The term "simple predicate" is used by us to address those predicates, where no nested aggregation is used. In most other cases, *clasp* is often performing at least equally well, as we could see during our experiments.

[1] Minimum Shift Design

CHAPTER 7

Conclusion and Future Work

7.1 Summary

Due to the increasing globalization and a potentially difficult market situation, an effective and efficient organization of a company's workforce is becoming more and more important. This does not only hold for rapidly expanding multi-national corporations, but also for non-profit organizations, like hospitals, it can be crucial to use a well-structured and adequate schedule for the different groups of employees.

In our work, we investigated declarative logic programs for solving the Minimum Shift Design Problem, an important problem from the broad area of workforce scheduling. For this purpose, we proposed three modelling approaches, implemented in the paradigm of Answer Set Programming. Furthermore, existing benchmark instances were used for the experimental evaluation to allow a high degree of comparability with previous work.

In detail, the following parts were covered by this thesis:

- Modelling of real-world problems using Answer Set Programming
- Investigation of the performance characteristics of different solvers
- Comparing our modelling approaches with state-of-the-art algorithms

The first two of our approaches rely on the assumption that a perfect solution without any deviation from the actual requirements is possible. Since this is not the case in general instances of the Shift Design Problem, our third program is able to deal with those instances. For our work we used a set of tools from the Potsdam Answer Set Solving Collection as software environment to execute our programs.

During our experiments, both solvers (*clasp* and *unclasp*) we used were able to compute the optimal solution for almost all problem instances that have a solution without deviation from the requirements with the first two of our modelling approaches within one hour of execution time and many of them could be solved within the first five minutes.

Despite the fact that many parts of our third program are based on the two implementations mentioned beforehand, the time needed to find a solution with this approach is often much worse than expected. We assume that the significant degradation in terms of performance is caused by the higher number of optimization criteria in combination with a smaller amount of integrity constraints, which drastically increases the complexity, respectively the size of the search space that has to be investigated. This assumption was confirmed during our experimental evaluation when we compared our our modelling approaches. Nevertheless, these findings indicate that the performance of our third approach implemented in ASP is currently not competitive to previous work in literature.

Although we cannot solve the general case of the Shift Design Problem satisfactorily at the moment, we note that the solution methodology proposed in our first two programs could be very useful for further development in future work. In conclusion, ASP as a declarative programming language has been shown to be a elegant and highly maintainable approach for solving the Shift Design Problem, but we have to admit that there is still work to do in order to obtain a competitive and robust solver. Some ideas for improved implementations are provided in the following section.

7.2 Future Work

Solving the practical case of the Shift Design Problem, where deviation from the staff demand is allowed, still remains a challenge for ASP. Subsequently, we provide a small subset out of the variety of possible extensions and improvements for our approaches which can be considered for future work:

Refine model for general problem instances: The third approach has significant problems to handle larger real-world instances. Therefore a more adequate approach, built without relying too much on the first two programs, could be worth a try.

Increase scalability and robustness: At the moment, minimal changes to the requirements can lead to significant differences in the execution time needed to find the optimal solution. For instance, currently there is no direct way for the solvers to infer that selecting three workers implies selecting two workers. A more adequate model could help to overcome this problem.

Combine ASP with heuristics: After a desirable quality of the solution has been obtained via heuristics, like local search, ASP could be used to enumerate a set of schedules with equal fitness values like the original solution, since this behaviour can be implemented very easily in the paradigm of ASP.

The aforementioned idea of combining ASP with heuristics could be implemented by first computing the fitness value of the initial solution generated by an existing solver for the Shift Design Problem and afterwards, a logic program could be used to generate answer sets with equal or even better characteristics by setting appropriate constraints.

Another idea is to generate the domain of possible values for the number of workers using heuristics and then the exhaustive search strategy of ASP could be employed to find a global optimum with the given configuration.

Ultimately, this could lead to a new generation of *hybrid* solvers for the Shift Design Problem which combine the advantages of heuristics, such as the high robustness when facing huge search spaces, with the benefits of ASP, as there is for instance the capability to perform exhaustive search in an efficient and highly maintainable way.

APPENDIX A

ASP-Implementations for Shift Design

A.1 Program “Exact1”

```
1  #const timeslots =
2      days * timeslots_per_day.

3  day(0 .. days - 1).
4  timeslot(0 .. timeslots - 1).

5  change(0, Requirement1 - Requirement2) :-
6      required(0, Requirement1),
7      required(timeslots - 1, Requirement2).
8  change(Time, Requirement1 - Requirement2) :-
9      required(Time, Requirement1),
10     required(Time - 1, Requirement2).

11 length(MinLength .. MaxLength) :-
12     shift_length(_, Length, SlotsAfter, SlotsBefore),
13     MinLength = Length - SlotsBefore,
14     MaxLength = Length + SlotsAfter.

15 amount(0 .. MaxAmount) :-
16     MaxAmount = #max [ required(_, Requirement) = Requirement ].

17 min_requirement(Start, Length, Value) :-
18     possible_shift_start(_, Start, Length),
19     EndSlot = Start + Length - 1, EndSlot < timeslots,
20     Value = #min [
21         required(Start .. EndSlot, Requirement) = Requirement
22     ].
```

Program “Exact1”: Part 1

```
23 min_requirement(Start, Length, Value) :-
24     possible_shift_start(_, Start, Length),
25     EndSlot = Start + Length - 1, EndSlot >= timeslots,
26     Value = #min [
27         required(Start .. timeslots - 1, Requirement)
28             = Requirement,
29         required(0 .. EndSlot - timeslots, Requirement)
30             = Requirement
31     ].

32 possible_shift_start(Shift, Start, Length) :-
33     timeslot(Start), length(Length),
34     shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
35     shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
36     day(Day), Offset = Day * timeslots_per_day,
37     Start >= Start1 + Offset - SlotsBefore1,
38     Start <= Start1 + Offset + SlotsAfter1,
39     Length >= Length1 - SlotsBefore2,
40     Length <= Length + SlotsAfter2,
41     required(Start + Length - 1, Requirement1),
42     Requirement1 > 0.

43 possible_shift_start(Shift, Start, Length) :-
44     timeslot(Start), length(Length),
45     shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
46     shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
47     day(Day), Offset = Day * timeslots_per_day,
48     Start >= Start1 + Offset - SlotsBefore1,
49     Start <= Start1 + Offset + SlotsAfter1,
50     Length >= Length1 - SlotsBefore2,
51     Length <= Length + SlotsAfter2,
52     required(Start + Length - timeslots - 1, Requirement1),
53     Requirement1 > 0.

54 Requirement [
55     assigned(Start, Length, Amount) = Amount :
56         min_requirement(Start, Length, Requirement1) :
57         amount(Amount) :
58         Start <= Time : Start + Length - 1 >= Time :
59         Amount <= Requirement1,
60     assigned(Start, Length, Amount) = Amount :
61         min_requirement(Start, Length, Requirement1) :
62         amount(Amount) :
63         Start + Length - timeslots - 1 >= Time :
64         Amount <= Requirement1
65 ] Requirement :- required(Time, Requirement).
```

Program "Exact1": Part 2

```
66 selected_shift(Start, Length) :-
67     possible_shift_start(Shift, Start1, Length), day(Day),
68     assigned(Start1, Length, Amount), timeslot(Start),
69     Start = Start1 - Day * timeslots_per_day,
70     Start < timeslots_per_day, Amount > 0.

71 :- assigned(Start, Length, Amount1),
72    assigned(Start, Length, Amount2),
73    Amount1 > Amount2.

74 :- change(Time, Change),
75    not Change [
76        assigned(Time, _, Amount) = Amount :
77            amount(Amount),
78        assigned(Start, Length, Amount) = -Amount :
79            amount(Amount) : Start + Length == Time,
80        assigned(Start, Length, Amount) = -Amount :
81            amount(Amount) : Start + Length - timeslots == Time
82    ] Change.

83 :- optimal_shift_count(OptimalCount),
84    not OptimalCount { selected_shift(_, _) } OptimalCount.
```

Program "Exact1": Part 3

A.2 Program "Exact2"

```
1  #const timeslots =
2      days * timeslots_per_day.

3  day(0 .. days - 1).
4  timeslot(0 .. timeslots - 1).

5  change(0, Requirement1 - Requirement2) :-
6      required(0, Requirement1),
7      required(timeslots - 1, Requirement2).
8  change(Time, Requirement1 - Requirement2) :-
9      required(Time, Requirement1),
10     required(Time - 1, Requirement2).

11 length(MinLength .. MaxLength) :-
12     shift_length(_, Length, SlotsAfter, SlotsBefore),
13     MinLength = Length - SlotsBefore,
14     MaxLength = Length + SlotsAfter.
```

Program "Exact2": Part 1

```
amount(0 .. MaxAmount) :-
    MaxAmount = #max [ required(_, Requirement) = Requirement ].

min_requirement(Start, Length, Value) :-
    possible_shift_start(_, Start, Length),
    EndSlot = Start + Length - 1, EndSlot < timeslots,
    Value = #min [
        required(Start .. EndSlot, Requirement) = Requirement
    ].

min_requirement(Start, Length, Value) :-
    possible_shift_start(_, Start, Length),
    EndSlot = Start + Length - 1, EndSlot >= timeslots,
    Value = #min [
        required(Start .. timeslots - 1, Requirement)
            = Requirement,
        required(0 .. EndSlot - timeslots, Requirement)
            = Requirement
    ].

possible_shift_start(Shift, Start, Length) :-
    timeslot(Start), length(Length),
    shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
    shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
    day(Day), Offset = Day * timeslots_per_day,
    Start >= Start1 + Offset - SlotsBefore1,
    Start <= Start1 + Offset + SlotsAfter1,
    Length >= Length1 - SlotsBefore2,
    Length <= Length + SlotsAfter2,
    required(Start + Length - 1, Requirement1),
    Requirement1 > 0.

possible_shift_start(Shift, Start, Length) :-
    timeslot(Start), length(Length),
    shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
    shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
    day(Day), Offset = Day * timeslots_per_day,
    Start >= Start1 + Offset - SlotsBefore1,
    Start <= Start1 + Offset + SlotsAfter1,
    Length >= Length1 - SlotsBefore2,
    Length <= Length + SlotsAfter2,
    required(Start + Length - timeslots - 1, Requirement1),
    Requirement1 > 0.
```

Program "Exact2": Part 2

```
Requirement [
    assigned(Start, Length, Amount) = Amount :
        min_requirement(Start, Length, Requirement1) :
        amount(Amount) :
        Start <= Time : Start + Length - 1 >= Time :
        Amount <= Requirement1,
    assigned(Start, Length, Amount) = Amount :
        min_requirement(Start, Length, Requirement1) :
        amount(Amount) :
        Start + Length - timeslots - 1 >= Time :
        Amount <= Requirement1
] Requirement :- required(Time, Requirement).

selected_shift(Start, Length) :-
    possible_shift_start(Shift, Start1, Length), day(Day),
    assigned(Start1, Length, Amount), timeslot(Start),
    Start = Start1 - Day * timeslots_per_day,
    Start < timeslots_per_day, Amount > 0.

:- assigned(Start, Length, Amount1),
   assigned(Start, Length, Amount2),
   Amount1 > Amount2.

:- change(Time, Change),
   not Change [
       assigned(Time, _, Amount) = Amount :
           amount(Amount),
       assigned(Start, Length, Amount) = -Amount :
           amount(Amount) : Start + Length == Time,
       assigned(Start, Length, Amount) = -Amount :
           amount(Amount) : Start + Length - timeslots == Time
   ] Change.

#minimize { selected_shift(_, _) }.
```

Program "Exact2": Part 3

A.3 Program "Flexible"

```
#const timeslots =
    days * timeslots_per_day.

day(0 .. days - 1).
timeslot(0 .. timeslots - 1).
```

Program "Flexible": Part 1

```
length(MinLength .. MaxLength) :-
    shift_length(_, Length, SlotsAfter, SlotsBefore),
    MinLength = Length - SlotsBefore,
    MaxLength = Length + SlotsAfter.

max_requirement(Start, Length, Value) :-
    possible_shift_start(_, Start, Length),
    EndSlot = Start + Length - 1, EndSlot < timeslots,
    Value = #max [
        required(Start .. EndSlot, Requirement) = Requirement
    ].

max_requirement(Start, Length, Value) :-
    possible_shift_start(_, Start, Length),
    EndSlot = Start + Length - 1, EndSlot >= timeslots,
    Value = #max [
        required(Start .. timeslots - 1, Requirement)
            = Requirement,
        required(0 .. EndSlot - timeslots, Requirement)
            = Requirement
    ].

possible_shift_start(Shift, Start, Length) :-
    timeslot(Start), length(Length),
    shift_start(Shift, Start1, SlotsAfter1, SlotsBefore1),
    shift_length(Shift, Length1, SlotsAfter2, SlotsBefore2),
    day(Day), Offset = Day * timeslots_per_day,
    Start >= Start1 + Offset - SlotsBefore1,
    Start <= Start1 + Offset + SlotsAfter1,
    Length >= Length1 - SlotsBefore2,
    Length <= Length + SlotsAfter2.

amount(0 .. MaxAmount) :-
    MaxAmount = #max [ required(_, Requirement) = Requirement ].

1 {
    assigned(Start, Length, Amount) :
        amount(Amount) : Amount <= MaxAmount
  } 1 :-
    max_requirement(Start, Length, MaxAmount).

selected_shift(Start, Length) :-
    possible_shift_start(Shift, Start1, Length), day(Day),
    assigned(Start1, Length, Amount), timeslot(Start),
    Start = Start1 - Day * timeslots_per_day,
    Start < timeslots_per_day, Amount > 0.
```

Program "Flexible": Part 2

```
timeslot_cover_value(Time, Value) :-
    required(Time, Requirement),
    Value = [ assigned(Start, Length, Amount) = Amount :
                  Start <= Time : Start + Length - 1 >= Time,
              assigned(Start, Length, Amount) = Amount :
                  Time <= Start + Length - timeslots - 1 ],
    Value >= Requirement - ucover_tolerance,
    Value <= Requirement + ocover_tolerance.

:- required(Time, _),
   0 { timeslot_cover_value(Time, _) } 0.

#maximize [
    timeslot_cover_value(Time, Value)
        = (Value - Requirement) @ 1 :
        required(Time, Requirement) :
        Value - Requirement < 0
    ].

#minimize [
    timeslot_cover_value(Time, Value)
        = (Value - Requirement) @ 1 :
        required(Time, Requirement) :
        Value - Requirement > 0
    ].

#minimize [ selected_shift(_, _) = 1 @ 1 ].
```

Program "Flexible": Part 3

Bibliography

[1] B. Andres, B. Kaufmann, O. Mattheis, and T. Schaub. Unsatisfiability-based optimization in clasp. In A. Dovier and V. Santos Costa, editors, *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, volume 17, pages 212–221. Leibniz International Proceedings in Informatics (LIPIcs), 2012.

[2] T. Aykin. Optimal shift scheduling with multiple break windows. *Management Science*, 42(4), 1996.

[3] T. Aykin. A comparative evaluation of modeling approaches to the labor shift scheduling problem. *European Journal of Operational Research*, 125(2):381–397, 2000.

[4] M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The USA-Advisor: A Case Study in Answer Set Planning. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Logic Programming and Nonmotonic Reasoning*, volume 2173 of *Lecture Notes in Computer Science*, pages 439–442. Springer, 2001.

[5] C. Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, United States, 2003.

[6] S.E. Bechtold and L.E. Jacobs. Implicit modeling of flexible break assignments in optimal shift scheduling. *Management Science*, 36(11):1339–1351, 1990.

[7] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, December 2011.

[8] M.J. Brusco and L.W. Jacobs. A simulated annealing approach to the cyclic staff-scheduling problem. *Naval Research Logistics (NRL)*, 40(1):69–84, 1993.

[9] F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1997.

[10] M. Côté, B. Gendron, and L. Rousseau. Grammar-based integer programming models for multiactivity shift scheduling. *Management Science*, 57(1):151–163, 2011.

[11] G.B. Dantzig. A comment on Eddie's "traffic delays at toll booths". *Operations Research*, 2(3):339–341, 1954.

[12] J. Van den Bergh, J. Beliën, P. De Bruecker, E. Demeulemeester, and L. De Boeck. Personnel scheduling: A literature review. *European Journal of Operational Research*, 226(3):367–385, 2013.

[13] L. Di Gaspero, J. Gärtner, G. Kortsarz, N. Musliu, A. Schaerf, and W. Slany. The minimum shift design problem. *Annals of Operations Research*, 155:79–105, 2007.

[14] L. Di Gaspero, J. Gärtner, N. Musliu, A. Schaerf, W. Schafhauser, and W. Slany. A hybrid LS-CP solver for the shifts and breaks design problem. In *Hybrid Metaheuristics*, volume 6373 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2010.

[15] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In S. Steel and R. Alami, editors, *Recent Advances in AI Planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1997.

[16] T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.

[17] A.T. Ernst, H. Jiang, M. Krishnamoorthy, and D. Sier. Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*, 153(1):3–27, 2004.

[18] J. Gärtner, N. Musliu, and W. Slany. Rota: a research project on algorithms for workforce scheduling and shift design optimization. *AI Communications*, 14(2):83–92, April 2001.

[19] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.

[20] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Challenges in answer set solving. In M. Balduccini and T.C. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2011.

[21] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In J. Delgrande and W. Faber, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 345–351. Springer, 2011.

[22] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.

[23] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, United States, 1988. The MIT Press.

[24] G. Grasso, S. Iiritano, N. Leone, V. Lio, F. Ricca, and F. Scalise. An ASP-Based System for Team-Building in the Gioia-Tauro Seaport. In M. Carro and R. Peña, editors, *Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 40–42. Springer, 2010.

[25] G. Grasso, S. Iiritano, N. Leone, and F. Ricca. Some DLV Applications for Knowledge Management. In E. Erdem, F. Lin, and T. Schaub, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 5753 of *Lecture Notes in Computer Science*, pages 591–597. Springer, 2009.

[26] W.B. Henderson and W.L. Berry. Heuristic methods for telephone operator shift scheduling: An experimental analysis. *Management Science*, 22(12):1372–1380, 1976.

[27] H. Ishebabi, P. Mahr, C. Bobda, M. Gebser, and T. Schaub. Answer set versus integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. *International Journal of Reconfigurable Computing*, 2009:6:1–6:11, 2009.

[28] H.C. Lau and S.C. Lua. Efficient multi-skill crew rostering via constrained sets. In *Proceedings of the Second ILOG Solver and Scheduler Users Conference*, pages 383–396, 1997.

[29] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.

[30] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1–2):39–54, 2002.

[31] V. Lifschitz. What is answer set programming? In D. Fox and C.P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, pages 1594–1597. AAAI Press, 2008.

[32] V.W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In K. Apt, V.W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer, 1999.

[33] S.L. Moondra. An LP model for work force scheduling for banks. *Bank Research*, 7(4):299–301, 1976.

[34] N. Musliu. *Intelligent Search Methods for Workforce Scheduling: New Ideas and Practical Applications*. PhD thesis, Technische Universität Wien, 2001.

[35] N. Musliu, A. Schaerf, and W. Slany. Local search for shift design. *European Journal of Operational Research*, 153(1):51–64, 2004.

[36] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[37] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An a-prolog decision support system for the space shuttle. In *Practical Aspects of Declarative Languages*, PADL '01, pages 169–183, London, United Kingdom, 2001. Springer.

[38] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Practical Aspects of Declarative Languages*, PADL '99, pages 305–319, London, United Kingdom, 1998. Springer.

[39] T. Syrjänen and I. Niemelä. The smodels system. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Logic Programming and Nonmotonic Reasoning*, volume 2173 of *Lecture Notes in Computer Science*, pages 434–438. Springer, 2001.

[40] G.M. Thompson. Improved implicit optimal modeling of the labor shift scheduling problem. *Management Science*, 41(4):595–607, 1995.

[41] G.M. Thompson. A simulated-annealing heuristic for shift scheduling using non-continuously available employees. *Computers & Operations Research*, 23(3):275–288, 1996.

[42] E. Tsang and C. Voudouris. Fast local search and guided local search and their application to british telecom's workforce scheduling problem. *Operations Research Letters*, 20(3):119–127, 1997.

[43] T. Walsh. Exploiting constraints. In *Inductive Logic Programming*, volume 7207 of *Lecture Notes in Computer Science*, pages 7–13. Springer, 2012.