DISSERTATION

TEMPLE - A Domain Specific Language for Modeling and Solving Real-Life Staff Scheduling Problems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Priv.-Doz. Dipl.-Ing. Dr.techn. Nysret Musliu
Institut für Informationssysteme (184/2)
Abteilung für Datenbanken und Artificial Intelligence

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Werner Schafhauser
0 1 2 6 3 3 2
Schelleingasse 17/6, A-1040 Wien

Wien, am 18. Oktober 2010

Kurzfassung

Ziel von Personalplanungsproblemen ist es, Dienstpläne zu erstellen, damit Unternehmen den Bedarf nach ihren Produkten und Dienstleistungen unter Einhaltung arbeitsrechtlicher Bestimmungen erfüllen können. Optimale oder nahezu optimale Lösungen für Personalplanungsprobleme verbessern die Arbeitsbedingungen für Mitarbeiter und helfen Betrieben, ihr Personal effizient und kostensparend einzusetzen. Unglücklicherweise sind Personalplanungsprobleme im Allgemeinen NP-hart und können daher nicht in polynomieller Zeit gelöst werden. Das Entwickeln guter Algorithmen für Personalplanungsaufgaben ist eine Kunst für sich selbst, und normalerweise sind solche Algorithmen sehr stark auf eine bestimmte Problemstellung zugeschnitten. Diese stark angepassten Lösungen können in der Regel nur sehr schwer für andere Probleme wiederverwendet werden, da bereits wenige geringfügige Änderungen in einem Problem zu vielen gravierenden Abänderungen und Erweiterungen in einem stark angepassten Algorithmus führen.

Ziel dieser Dissertation ist es, eine Modellierungssprache zu entwickeln, mit der wir Personalplanungsprobleme auf sehr natürliche, einfache und intuitive Weise modellieren und lösen können. Infolgedessen werden neue Lösungen für Personalplanungsprobleme wesentlich schneller entwickelt und bereits existierende Lösungen viel einfacher geändert und erweitert.

Um dieses Ziel zu erreichen, betrachten wir zuerst zwei konkrete Personalplanungsprobleme. Das erste stammt aus einem Call Center, das zweite Problem betrachtet eine ähnliche Aufgabenstellung für Überwachungspersonal. Für diese beiden Probleme entwickeln wir zwei maßgeschneiderte Lokale Suche Algorithmen, die in vertretbarer Zeit qualitativ hochwertige Lösungen erzielen. Basierend auf diesen beiden Lösungen identifizieren wir grundlegende Bestandteile von Personalplanungsproblemen und Lokale Suche Algorithmen und entwickeln die Modellierungssprache TEMPLE, in welcher diese Bestandteile realisiert werden. Des Weiteren implementieren wir einen TEMPLE-Übersetzer mit dessen Hilfe wir für jedes Problemmodell in TEMPLE drei Lokale Suche Algorithmen erzeugen können. Diese Algorithmen können sofort, ohne weitere Nutzereingabe auf ein konkretes Problem

angewandt werden, um optimierte Lösungen zu erzeugen.

Wir demonstrieren die Zweckmäßigkeit unseres Ansatzes, indem wir das Personalplanungsproblem aus dem Bereich Überwachungspersonal in TEMPLE modellieren und lösen, und zeigen, dass wir mit unserem Ansatz konkurrenzfähige Ergebnisse erzielen. Abschließend modellieren wir ein vielschichtiges Personalplanungsproblem in TEMPLE, in welchem wir zuerst ein legales, hinsichtlich des Personalbedarf optimiertes Pausenmuster berechnen, und anschließend konkrete Arbeitsaufgaben den einzelnen Mitarbeitern unter Berücksichtigung ihrer Qualifikationen zuweisen. Die Lokale Suche Algorithmen, die wir aus unserem TEMPLE Modellen generiert haben, sind Bestandteil eines kommerziellen Personalplanungstools, das bereits erfolgreich kundenseitig eingesetzt wird.

Abstract

Staff scheduling is the process of creating work timetables for personnel so that companies can satisfy the demands for their goods and services. Optimal or close to optimal solutions for staff scheduling tasks help companies to deploy their staff efficiently and cost savingly, and improve the working conditions for deployed staff as well. Unfortunately, staff scheduling problems are NP-hard in general, thus, they cannot be solved to optimality in polynomial time. As a consequence, the design of solutions for staff scheduling problems is an art in itself and results in algorithms which are strongly customized to a specific staff scheduling task. Customized solutions are usually very difficult to adapt, extend, and reuse for other problems.

In this thesis we develop a modeling language to formulate and solve staff scheduling tasks in a very natural, simple, and intuitive manner. Consequently, new algorithms for staff scheduling problems can be obtained more quickly, and already existing solutions can be modified and extended easily.

To achieve that goal, we first consider two real-life staff scheduling problems, one originating in a call center, the other arising from the area of supervisory personnel, and we develop two customized local search algorithms, which are able to generate high-quality solutions for the two problems in reasonable time. On the basis of these two customized solutions we abstract common features and basic building blocks of staff scheduling problems and local search techniques. Then, to model staff scheduling problems with reduced effort, we design a novel, domain specific language called TEMPLE. Furthermore, we develop and implement a TEMPLE compiler which transforms TEMPLE models of staff scheduling tasks into three generic local search algorithms, which can be executed instantaneously to optimize a considered staff scheduling problem.

To deliver a proof of concept, we model the staff scheduling problem for supervisory personnel in TEMPLE, and we show that our approach is able to achieve competitive results of acceptable quality in reasonable time. Finally, we model and solve a multilayered, real-life break scheduling and task assignment problem in TEMPLE. The local search algorithms obtained from our TEMPLE model represent the core of a commercial staff scheduling tool which is already used successfully on customer's site.

Acknowledgments

First and foremost I would like to thank my supervisor Nysret Musliu for the excellent support he gave me while pursuing my PhD. Due to his very nice character and his readiness to listen and help me at any time writing this thesis was indeed a great pleasure for me.

My thanks goes also to Johannes Gärtner and Wolfgang Slany for their discussions within our research project which opened my mind to novel ideas and approaches. Furthermore, I would like to thank Sabine Wahl, Ruth Sigär, and Karin Boonstra-Hörwein, for the perfect cooperation we had while developing creative solutions for complex staff scheduling tasks. Moreover, I say thanks to Michael Jakl for his expertise on ANTLR as well as to Toni Pisjak for the technical support he gave me during the last years. Finally, I would like to thank Pascal Van Hentenryck, Laurent Michel, and the Dynadec Support Team, for answering my questions and satisfying my requests concerning the Comet optimization language.

The research herein was partially conducted within the competence network Softnet Austria (`http://www.soft-net.at/`) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of staff scheduling problems is the creation of work timetables in order that companies can satisfy the demands for their goods and services. Staff scheduling is a very complex process, encompassing several different phases, such as determining staffing requirements, constructing shift plans and break schedules, creating rosters for individual employees, and assigning tasks or services to be performed. In each single step we must obtain solutions which on the one hand are consistent with legal requirements and labor regulations, and on the other hand deploy staff efficiently. Thus, optimal or close to optimal solutions for staff scheduling problems improve the working conditions for employees, and help companies to deploy their personnel cost savingly. Unsurprisingly, staff scheduling problems are of high practical relevance and represent hot topics of basic as well as applied research.

Unfortunately, many staff scheduling problems are NP-hard, and as a consequence, they cannot be solved to optimality in polynomial time. Therefore, staff scheduling problems are solved by using mathematical programming, or sophisticated AI-techniques, such as constraint programming, heuristics, meta-heuristic methods, or branch and bound algorithms. No matter which of these approaches we follow, the design of algorithms for staff scheduling problems is an art in itself and we end up with a solution that is strongly customized to a specific task.

Strongly customized solutions are usually very difficult to adapt, extend and maintain. A few minor changes within a problem's specification can result in many major modifications or a completely new implementation of a customized algorithm, and while developing solutions for staff scheduling problems, changes will occur constantly and inevitably, for the following reasons:

- ▷ The very same staff scheduling problem will never occur in two or several companies. Even though two companies deal with a similar staff scheduling problem,

each company has its own, specific criteria which have to be considered. Often, these special requests are the direct result of negotiations between the management board and the workers' council, or they reflect working conditions prevailing in a particular industry.

- ▷ An exact problem specification does not exist in advance, instead the problem specification is developed and evolved together with the solution. Due to the multilayered nature of staff scheduling problems, it is not possible to define all criteria involved in a specific problem at the beginning of the software engineering process. Thus, it is advisable to develop solutions for staff scheduling problems by following modern, agile software engineering paradigms. Agile software development approaches are characterized by many short development cycles, between which additional criteria or modifications requested by users can be realized.

To sum up, the reasons why staff scheduling problems represent very interesting tasks are their high practical relevance, their general NP-hardness, and the great efforts associated with the development of effective, customized solutions.

1.1 Research Questions of This Thesis

The main intention of this thesis is to model and solve real-life staff scheduling problems at reduced development effort. To achieve that goal we have to answer the following research questions:

- ▷ We want to develop a domain specific language to model staff scheduling problems in a very natural, simple, and intuitive manner. By the help of these techniques, new software solutions for staff scheduling problems shall be obtained more quickly, and already existing solutions shall be modified and extended easily.
- ▷ With the domain specific language not only we want to model staff scheduling problems effectively but also we wish to solve them efficiently. For that purpose, we desire to design and implement a solver for our domain specific language which optimizes staff scheduling problems by the help of generic local search algorithms.
- ▷ The proposed domain specific modeling language shall hide any domain-specific knowledge of staff scheduling tasks or optimization algorithms. In that manner they can be used by any ordinary developer or end-user, who must not be necessarily a domain-expert.

- ▷ We want to deliver a proof of concept that our domain specific modeling language can be applied successfully to real-life staff scheduling problems in practice.

1.2 Main Results of This Thesis

To answer the previously mentioned questions within this thesis, we first consider two real-life staff scheduling problems. The first problem is a real-life break scheduling problem for supervisory personnel, the second one is another break scheduling problem originating from a call center. In both problems we are given staffing requirements and already designed shift plans, and we must schedule breaks within these shift plans in order that several constraints concerning the legality of break times and break patterns are satisfied, and shortage of staff is reduced to a minimum degree. For these two tasks we develop two customized local search algorithms and we successfully obtain solutions of acceptable quality within acceptable time.

On the basis of these two specific tasks we identify common features and basic building blocks of general staff scheduling problems, such as time intervals, links between time intervals, and curves. Furthermore, we observe, that in staff scheduling tasks, properties, curves, and constraints, can be derived from each other, step by step, in a modular manner. Thus, we require from a domain specific modeling language for staff scheduling problems, to provide abstractions and notations reflecting these basic building blocks and to enable a stepwise formulation of properties and constraints. Moreover, since we want to solve staff scheduling problems via local search techniques, a domain specific language shall also incorporate essential components of local search algorithms, i.e., initial solutions, objective functions, and moves.

We review state-of-the-art modeling languages [56, 22, 23, 31, 32, 38] and metaheuristic frameworks [9, 27, 30], and we examine whether basic building blocks of staff scheduling problems and local search algorithms are offered by these approaches. In neither of the considered modeling languages staff scheduling problems can be modeled as we desire it to do, either they are aimed at general combinatorial optimization problems and do not support basic building blocks of staff scheduling tasks, or they are targeted at a slightly different problem domain. As to metaheuristic frameworks, they require from potential user detailed knowledge of a framework's structure, of object orientated programming, and of local search techniques, thus, they are less suited to be applied by non-domain-experts. Therefore, we decide to develop TEMPLE, a novel domain specific language for modeling and solving staff scheduling problems at reduced effort. With TEMPLE we achieve the following design goals:

Modularity: A TEMPLE model consists of small, concise building blocks reflecting common features of staff scheduling problems. New building blocks are derived

from already existing ones. By this principle users are forced to formulate a complex problem in small, concise and traceable steps. Consequently, the resulting problem models are well-structured, easy to understand, modify and maintain.

Adaptability and Extensibility: Problems modeled in TEMPLE can be adapted easily. A few small changes in the problem formulation result only in a few small changes in the corresponding TEMPLE program.

Simplicity: TEMPLE requires only basic programming skills from its users. Anybody familiar with a third generation programming language should be able to understand and use TEMPLE.

Automatic Optimization: Once a problem is modeled in TEMPLE it can be optimized in an instant without requiring additional coding from the user.

Openness: In contrast to other constraint-based modeling languages, TEMPLE is not restricted to a finite set of predefined features or constraints. We can model arbitrary features or constraints of staff scheduling problems in TEMPLE.

Efficiency: TEMPLE's intrinsic computational overhead is kept as little as possible. Thereby we ensure that problems cannot only be modeled effectively but also solved efficiently with TEMPLE.

To enable automatic optimization of staff scheduling problems we develop a TEMPLE compiler which transforms the TEMPLE model of a staff scheduling problem into three executable local search algorithms: a simulated annealing algorithm [36], a hill-climbing based approach [39], and an iterated local search algorithm [37]. These algorithms can be applied instantaneously and do not require any further user input or modifications.

The key idea behind local search techniques is to repeatedly apply small changes to intermediate solutions in order to find higher-quality solutions. In each step, local search techniques examine solutions closely related to the current one, a so-called local neighborhood, and select one solution within that local neighborhood to be the next current solution. Usually the local neighborhood is computed by applying small changes, also denoted as moves, to the current solution. Although there are significant differences among the local search algorithms generated by the TEMPLE compiler they basically apply the same three main steps in each iteration:

1. They compute a set of moves to obtain a local neighborhood of the current solution.

2. They evaluate the effect of each move on the current solution. When evaluating a move they check whether the move is feasible, i.e., it does not cause any hard constraint violations, and they determine the difference in the problem's objective function resulting from the move.

3. They select a feasible move and execute it to obtain a new solution.

In local search algorithms, most computational effort is spent on the evaluation and execution of moves. To ensure that these two steps are carried out efficiently, we apply the following strategies in the local search algorithms created by the TEMPLE compiler:

Lazy Evaluation: If we observe that a move violates a hard constraint we will not evaluate the move's effect on other hard and soft constraints.

Caching: We use a move cache to store the result of each evaluation of a move on a property, curve or constraint. With that move cache we can avoid that a move is evaluated several times for the same derived element.

Efficient Move Evaluation and Execution: When evaluating a move's effect on a solution we only evaluate those properties, curves and constraints that are affected by the move. Similarly, when performing a move we update only those solution elements which are actually changed by a move.

Efficient Data Structures: To evaluate a move's effect on curves efficiently we developed and implemented a speed-up strategy. This strategy ensures that only those curve positions are evaluated which are affected by a move. By applying that speed-up strategy we could reduce the computational costs associated with curves significantly.

To demonstrate TEMPLE's modularity, simplicity, and openness, we reconsider the real-life break scheduling problem for supervisory personnel, and model it in TEMPLE. The resulting TEMPLE program consists of only 500 lines of code and is written in a very concise, understandable and modular manner. In total we needed one man-week to develop a suitable TEMPLE model for the considered break scheduling problem. An experimental evaluation of the iterated local search algorithm on real-life and randomly generated benchmark instances reveals that TEMPLE is able to compute solutions of high quality in acceptable time.

Finally, we consider a multilayered break scheduling and task assignment problem. In this staff scheduling problem we are given task requirements for an entire day, an already existing shift plan and the qualifications of each employee. To obtain a solution we must compute a break schedule which is completely consistent with a set of legal

requirements and labor regulations, and in addition, we must also assign the required tasks to available employees in accordance with their qualifications. Furthermore, task assignments must satisfy several criteria. For instance, each task must be performed by the same employee for a certain number of minutes and employees should not be forced to change the task they carry out.

Since the considered problem is very complex as a whole we decompose it into three separate phases each of which is modeled and solved by a separate TEMPLE program. In the first phase we compute a break schedule which is consistent with all legal requirements. In the second phase we optimize the break schedule with respect to the task requirements. In the third phase we assign the required tasks to available employees and we optimize the task assignment with respect to the imposed criteria.

For the considered break scheduling and task assignment problem, we extend and adapt the TEMPLE program for the break scheduling problem for supervisory personnel to solve the first and the second phase of the decomposed problem. By modeling tasks as intervals linked with employees we also succeed in modeling phase three in a very natural way.

The three resulting TEMPLE models represent the core of a break scheduling and task assignment tool. With a prototype of that tool we deliver a proof of concept that automated break scheduling and task assignment was possible within a reasonable amount of time, i.e., approximately five minutes on a state of the art computer. The prototype has been extended into a commercial application, which is already used successfully by decision makers in their day-to-day business.

To sum up, at this point we state explicitly the main results achieved within this thesis:

- ▷ We develop and implement a min-conflicts based algorithm for a real-life break scheduling problem for supervisory personnel. Thanks to that algorithm we can generate high-quality solutions that fulfill labor rules and legal requirements, and at the same satisfy staffing demands.
- ▷ We adapt the min-conflicts based algorithm for a related problem arising in a real-life call center. Computational results on randomly generated benchmark instances reveal that the modified algorithm is able to create close to optimal solutions within reasonable time. The min-conflicts based algorithm is applied successfully at the call center where it computes the daily break schedules for call center agents.
- ▷ We propose TEMPLE, a novel domain specific language designed to model staff scheduling problems in a very modular and natural manner. For that purpose we identify common features and basic building blocks of staff scheduling tasks and

local search techniques which are reflected by the abstractions, notations, and language elements offered in the TEMPLE language. Thanks to TEMPLE, ordinary developer and end users can model staff scheduling problems more concisely, and the obtained programs can be easily modified and adapted to similar staff scheduling tasks.

- ▷ We develop and implement a TEMPLE compiler transforming TEMPLE models of staff scheduling problems into three executable local search algorithms. These algorithms can be applied instantaneously and do not require any further user input or modifications. To ensure that the obtained algorithms are carried out efficiently, we implement several strategies within the compiler, in order that only as many computations as necessary are performed.

- ▷ We reconsider the break scheduling problem for supervisory personnel and model it with TEMPLE. We show that the obtained model is indeed built in a modular manner by using small concise building blocks, and that the iterated local search algorithms generated by our TEMPLE compiler is able to produce competitive results of acceptable quality in reasonable time.

- ▷ We present a multilayered, real-life break scheduling and task assignment problem, and solve it with TEMPLE. Thereby we deliver a proof of concept that TEMPLE can be applied successfully to real-life staff scheduling problems. The local search algorithms obtained from our TEMPLE model represent the core of a commercial staff scheduling tool which is currently used successfully on customer's site to generate daily break schedules and task assignments and to react on intraday changes.

The results presented in Chapter 3 and Chapter 4 of this thesis have already been published in the journal "IEEE Intelligent Systems" and in the proceedings of the "7th International Conference for the Practice and Theory of Automated Timetabling". In addition, some of the ideas and figures described in Chapter 5 through Chapter 8 have been submitted to the "26th ACM Symposium on Applied Computing (SAC)" and the journal "Engineering Applications of Artificial Intelligence".

1.3 Further Organization of This Thesis

Chapter 2 gives preliminary information about staff scheduling problems and local search algorithms. In Chapter 3 we address a real-life staff scheduling problem for supervisory personnel and solve it with a min-conflicts-based local search strategy. Chapter 4 presents a similar problem originating from a call center. In Chapter 5 we identify

common features and basic building blocks of staff scheduling problems and design the domain specific modeling language TEMPLE. Chapter 6 reviews related modeling languages as well as metaheuristic frameworks and compares these approaches with TEMPLE. In Chapter 7 we design and implement a TEMPLE compiler transforming TEMPLE models into three generic local search algorithms which can be executed instantaneously to solve the underlying staff scheduling tasks. In Chapter 8 we model and solve two real-life staff scheduling problems, and we report on the application of TEMPLE within a commercial staff scheduling software. Finally, Chapter 9 concludes and describes future work.

Chapter 2

Preliminaries

2.1 A Classification of Staff Scheduling Problems

The goal of staff scheduling problems is the creation of work timetables for personnel so that companies can satisfy legal requirements as well as the demands for their goods and services. Staff scheduling is a multilayered process, usually consisting of several different subtasks, each of which representing a complex problem taken by itself. In the following, we will give an overview of the different facets of the overall staff scheduling process, whereby we refer to a classification given by Ernst et al. [19].

2.1.1 Demand Modeling

Demand modeling is the process of determining or estimating how many staff must be deployed over a certain planning period. The staff are needed to perform activities, tasks or services during that time, thus, we have to assess which activities, tasks and services are required to be performed. Then we must derive the number of employees required to carry out these duties. As a result we obtain staffing requirements over a planning period, specifying for each time point the number of employees that should be working at that time. Ernst et al. [19] mention three different categories of demand modeling problems which frequently occur in practice:

Task based demand modeling. Staffing requirements are derived directly from a series of individual tasks to be performed. Task based demand modeling is usually applied in areas where tasks and the exact times when they have to be performed are known in advance, like in transport applications or product assembly. Figure 2.1 (a) sketches how staffing requirements are obtained via task based modeling.

Flexible demand modeling. In that case activities, tasks or services to be performed are not known exactly beforehand, thus, they must be estimated with forecasting techniques. For instance, flexible demand modeling is frequently applied to obtain the staffing requirements for call centers, retail stores or airport check-in counters.

Shift based demand modeling. Above all, shift based demand modeling is applied within the health care sector, where staffing requirements can be derived directly from a specification of the number of staff that are required to be on duty during different shifts.

2.1.2 Shift and Break Scheduling

Typically, in shift scheduling problems we generate shift plans in accordance with given staffing requirements. For that purpose we must decide what kinds of shifts are used within a shift plan and we must determine the number of employees that should be working in each single shift on each single day. Good shift plans satisfy staffing requirements, thereby they avoid shortage and excess of staff and guarantee that required activities, tasks or services can be carried out effectively, and personnel are deployed cost-savingly.

As a further aspect of shift scheduling problems, we must often schedule work and meal breaks for shifts or single employees. The obtained break schedules shall be consistent with legal requirements, labor rules resulting from agreements between the labor council and the management board, and ergonomic criteria. Break scheduling tasks arise pre-eminently in working areas where employees spend their working time in front of computer monitors, such as call centers. The combined problem of scheduling shifts and breaks at the same time represents a very complex task, thus, the entire problem is often solved in two phases. In the first step a shift plan without any breaks is constructed, and then, breaks are inserted in a subsequent phase.

Figure 2.1 (b) presents the staffing requirements and a (suboptimal) solution for a sample shift and break scheduling problem. In Figure 2.1 (b) we see that shortage and excess of staff still occur at some periods of time.

2.1.3 Line of Work Construction

After a shift plan has been obtained we must combine single shifts or duties to single lines of work. A line of work represents the shift or duty pattern for a single, individual employee. Also when constructing lines of work, we have to satisfy legal requirements and ergonomic criteria, e.g., in a feasible line of work, a night shift must not be followed immediately by a day shift. Figure 2.1 (c) shows three feasible lines of work constructed for an entire week.

2.1.4 Staff Assignment

During the process of staff assignment we assign individual employees to single lines of work (see Figure 2.1 (d)). Line of work construction and staff assignment are often solved together as a single staff scheduling problem.

2.1.5 Task Assignment

In task assignment problems we must assign activities or tasks required to be carried out to shifts or individual staff (see Figure 2.1 (e)). Thereby, we often have to consider the qualifications of employees and obtain a task assignment which is consistent with staff skills.

2.1.6 Further Surveys of Staff Scheduling Problems

Besides the article on general staff scheduling and rostering by Ernst et al. [19] there exist several surveys of staff scheduling problems focussed on specific industries. Burke et al. [8] survey problems and solution approaches in the field of nurse rostering. A further overview of different models and methodologies for nurse rostering is given by Cheang et al. [14]. Arabeyre et al. [2] and Barnhart et al. [5] present a review of the literature, problems, and solutions techniques in the area of airline crew scheduling. Considering railway crew scheduling problems, many effective models and solution approaches have been proposed by Caprara et al. [12, 13, 11]. The reader is referred to Aksin [1] and Ernst [18] for an overview of different staff scheduling tasks and problems appearing in the modern call center industry.

2.2 State-of-the-Art in Break Scheduling Problems

The staff scheduling problems considered in this thesis are largely break scheduling problems. Thus, at this point, we give a short review on previous work performed in the area of break scheduling.

Break scheduling has been mainly considered in the literature as part of the shift scheduling problem. Dantzig developed the original set-covering formulation [15]. In this formulation there exists a variable for each feasible shift. Feasible shifts are enumerated based on possible shift starts, shift lengths, breaks, and time windows for breaks. When the number of shifts increases rapidly, this formulation is not practical. Bechtold and Jacobs proposed a new integer programming model [6]. In their formulation, the modeling of break placements is done implicitly. Authors reported superior results with

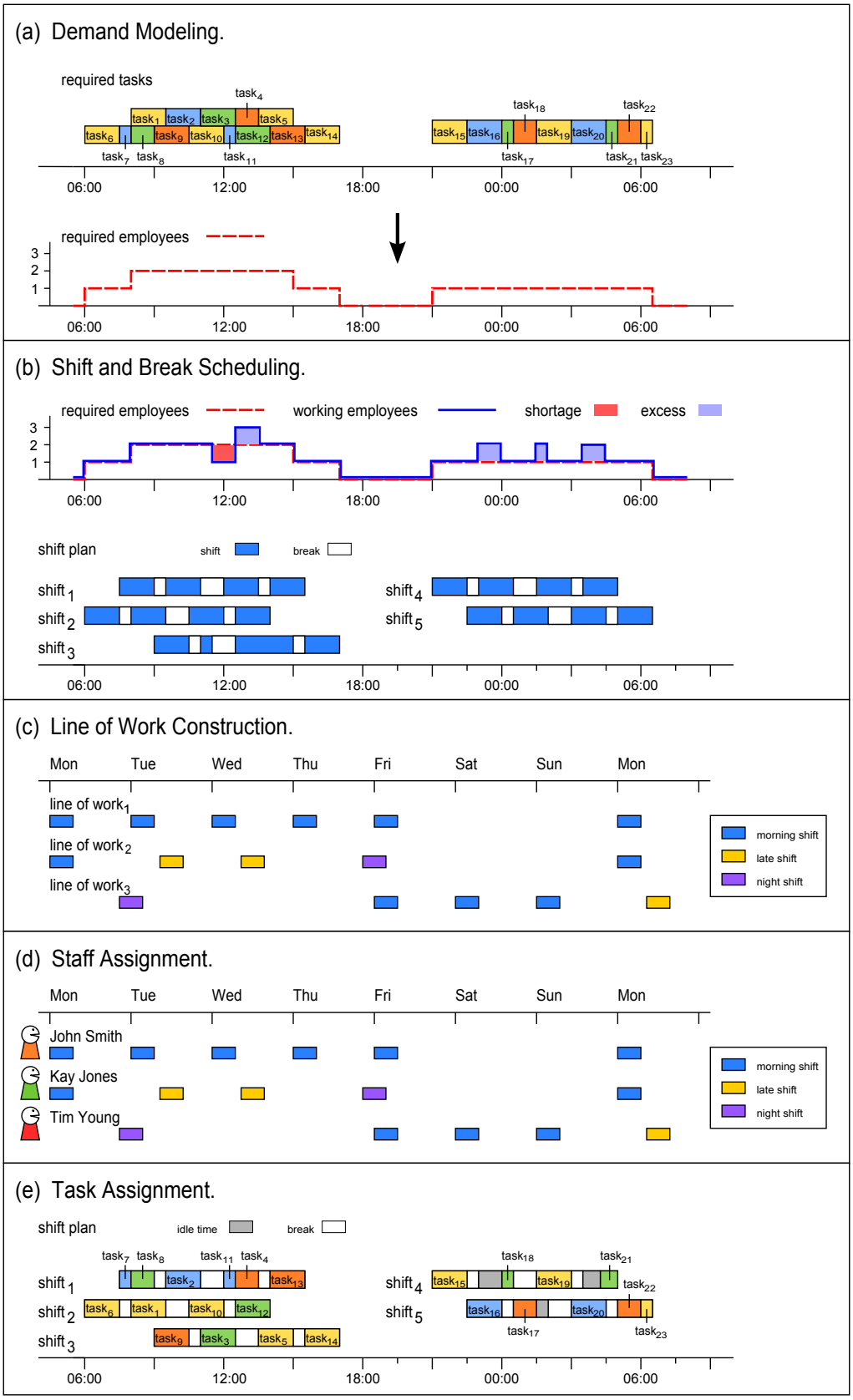


Figure 2.1: Subtasks involved in the overall staff scheduling process according to Ernst et al. [19].

their model compared to the set covering model. However, their approach is limited to scheduling problems of less than 24 hours per day. Thompson introduced a fully implicit formulation of the shift scheduling problem [53]. A comparison of different modeling approaches is given by Aykin [4]. Rekik et al. [48] developed two other implicit models and improved upon previous approaches including AykinŠs original [3] by up to about 10 percent.

A greedy algorithm for scheduling breaks after generating shifts has been presented in [25]. The authors propose a simple algorithm which includes the phase of assigning the breaks greedily based on the information for the largest excess, and then applying simple repair steps on the assigned breaks.

Tellier and White present a tabu search based approach in order to solve a scheduling problem originating from call centers [52]. They aim at minimizing the squared deviation of working employees from staffing requirements while various constraints are required to be satisfied. In [52] there is a correspondence between shifts and real employees of the contact center, and the constraints on a feasible solution restrict the position of breaks within shifts, the position and lengths of single shifts within the entire schedule, and the minimum and maximum number of paid working hours per employee. Canon investigates also the use of tabu search for the shift design problem including breaks [10].

Thompson and Pullman [54] argue that scheduling breaks simultaneously with shifts increases the quality of obtained shift plans. Although this is reasonable, in many real-life scenarios, like in our problem, it is required to just schedule breaks in already existing fixed shift plans.

The break scheduling problem we will adress in Chapter 3 has recently also been investigated in [44, 58, 59]. A memetic algorithm for this problem is presented in [44]. The moves applied in [44] are based on on our work, that will be shown in Chapter 3. Moreover, the memetic algorithms for that problem have been further improved by applying a new memetic representation and a penalty mechanism for memes [58, 59].

2.3 Local Search Algorithms

The great drawback in staff scheduling problems is that they are NP-hard in general. Examples of NP-hard staff scheduling problems are shift scheduling problems, such as the min-shift design problem of which even a logarithmic approximation is NP-hard [26], or break scheduling problems [58]. Due to their general NP-hardness, staff scheduling tasks cannot be solved to optimality in polynomial time at the present, and most likely not in future either. Therefore, staff scheduling problems are solved using mathematical

programming or sophisticated AI-techniques, such as constraint programming, heuristics, metaheuristics, or branch and bound algorithms.

Algorithm 1 Basic Local Search

1: **compute** an initial solution x
2: **repeat**
3: **select** a solution $x' \in N(x)$ within the local neighborhood of x
4: $x = x'$
5: **end select**
6: **until** *termination condition* is satisfied
7: **return** best solution found by the search algorithm

Local search techniques are a class of metaheuristic algorithms, which have been applied successfully to various staff scheduling tasks. The key idea behind local search algorithms is to repeatedly apply small changes to intermediate solutions in order to find higher-quality solutions. Algorithm 1 presents a the basic priciples of a local search algorithm in pseudo code notation. In the following we introduce and describe frequently used terms within the area of local search algorithms:

Search space. The search space is the set of all solutions for a particular optimization problem.

Objective function. An objective function, also referred to as fitness function, maps solutions within the search space to real values. The overall goal of a local search algorithm is to find a solution minimizing or maximizing the objective function of the considered problem.

Initial solution. A solution or point within the search space at which a local search algorithm starts.

Move. A move is a small change which is applied to obtain a further solution.

Local neighborhood. The local neighborhood of a current solution is a set of solutions closely related to the current one. A local neighborhood is obtained by applying different moves to the current solution.

Local optimum. A local optimum is a solution having a better objective value than any other solution within its local neighborhood. When constructing local search algorithms a crucial point is to develop and implement strategies to escape local optima and explore further regions within the search space.

Global optimum. A global optimum is an optimal solution of a considered optimization problem.

Figure 2.2 presents the search space of a fictitious maximization problem. The height associated with a point encodes the objective value of the corresponding solution. Further we see a possible execution of a local search algorithm: Starting at an initial solution, the algorithm visits solutions of improved objective value until a local optimum is reached. After escaping the local optimum the algorithm explores new regions of the search space, and finally, it reaches a globally optimal solution.

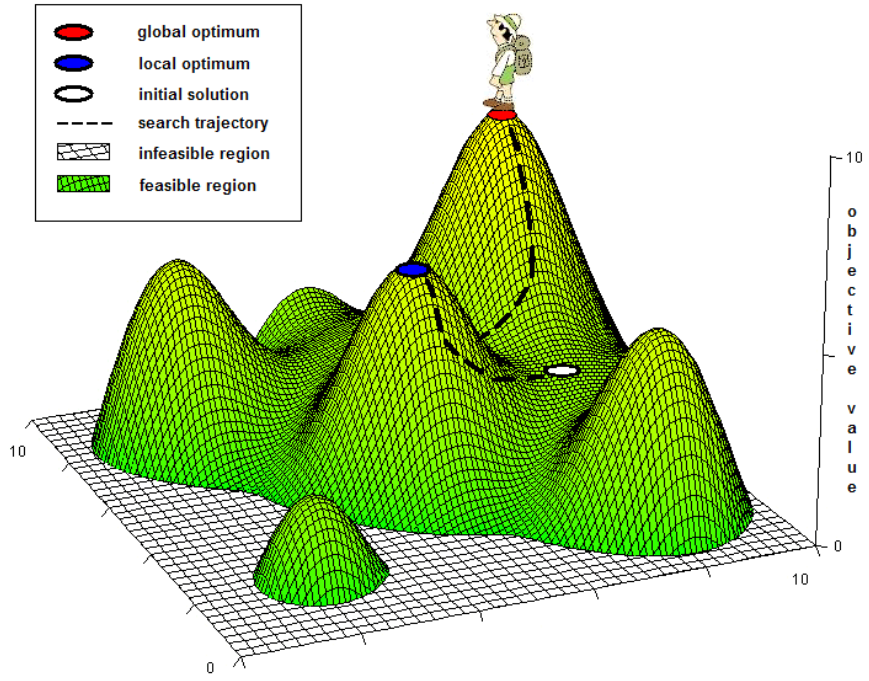


Figure 2.2: A fictitious execution of a local search algorithm in a two dimensional search space.

2.3.1 An Overview of Local Search Techniques Used in This Thesis

In this thesis we develop a domain specific language to model and solve staff scheduling problems. For that purpose we design and implement a compiler which transforms

problem models written in the domain specific language into three generic local search algorithms that can be executed instantaneously without requiring any further user input. In the following, we will describe the basic principles of these three local search algorithms in detail.

Hill Climbing with Random Noise

The first local search algorithm is presented in Algorithm 2 in pseudo code notation. [39]. As input the algorithm is given a small probability p_{noise} controlling the behaviour of the local search process. After an initial solution is obtained, the algorithm applies either a hill climbing strategy [39] with high probability $1 - p_{noise}$ or performs a random noise move with small probability p_{noise}. The hill climbing strategy computes a local neighborhood and selects the best feasible solution having a better objective function value. If the search reaches a local optimum, i.e., a solution better than any other solution within that solution's local neighborhood, hill climbing will not proceed any further, since it will not find solutions of better quality. To escape local optima, we introduce the local noise process, which accepts any feasible solution in a local neighborhood, even a solution of minor quality.

Iterated Local Search

Iterated local search algorithms [37] consist of three components, a local search strategy used to reach locally optimal solutions, a so-called pertubation mechanism which is applied to escape from local optima and an acceptance criterion deciding whether the search process will continue from the current solution or a previously obtained one. Figure 2.3 illustrates the interaction between local search and perturbation in a local search algorithm.

The iterated local search algorithm generated by our compiler is presented in Algorithm 3. First the iterated local search algorithm tries to reach a local optimum by applying the hill climbing strategy presented in Algorithm 4. The hill climbing algorithm terminates if no improvement has been achieved for H iterations and returns the obtained locally optimal solution x^*.

Then, the locally optimal solution x^* is passed to the pertubation process presented in Algorithm 5. The pertubation proceeds with any arbitrary feasible solution within a local neighborhood, and it terminates after P solutions of minor quality have been visited. Afterwards hill climbing is applied again until a new local optimum x'^* has been reached.

Finally, the algorithm decides whether the search should continue from the current local optimum x'^* or if the previously obtained local optimum x^* should be restored.

Algorithm 2 Hill Climbing with Random Noise(t, p_{noise}, s)

```
1:  compute an initial solution x
2:
3:  repeat
4:
5:    //hill climbing
6:    with probability 1 − p_noise do
7:      compute N(x, s) a local neighborhood of x of size s
8:      evaluate N(x, s)
9:      select the best feasible solution x' ∈ N(x, s)
10:       if x' < x then
11:         x = x'
12:       end if
13:     end select
14:   end with
15:
16:   //random noise
17:   with probability p_noise do
18:     select a random feasible solution x' in a local neighborhood of x
19:       x = x'
20:     end select
21:   end with
22:
23: until a time limit t has been reached or an optimal solution has been found
24: return best solution found by the search algorithm
```

In our local search algorithm we implemented the following three different acceptance criteria taking that decision:

Accept Always. The search continues with the last obtained local optimum x'^*.

Accept Best. The search continues with the better local optimum.

Accept Percentage. The search continues with solution x'^* if it is better or its loss of quality does not exceed a certain threshold.

Simulated Annealing

Simulated annealing [36] is based on an analogy from metallurgy. To grow crystals of high quality several substances are first melted and cooled in a controlled way after-

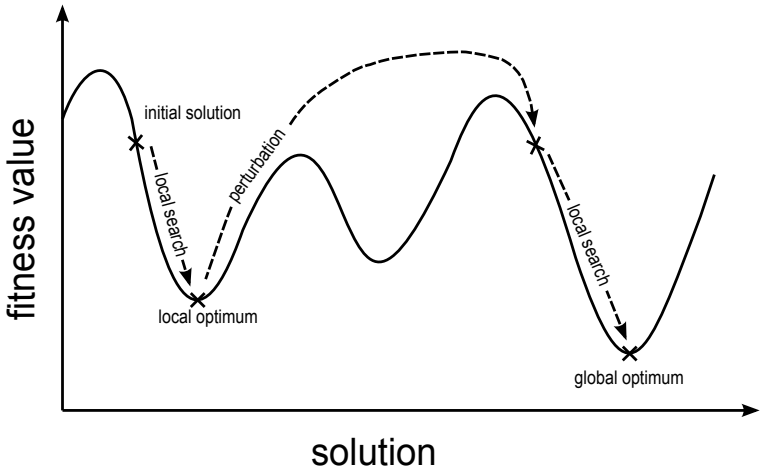


Figure 2.3: Interaction between local search and perturbation in an iterated local search algorithm.

Algorithm 3 Iterated Local Search(t, H, P, s)

1: **compute** an initial solution x_0
2: $x^* = Hill\ Climbing(x_0, H)$
3: **repeat**
4: $x' = Perturbation(x^*, P)$
5: $x'^* = Hill\ Climbing(x', H)$
6: $x^* = Acceptance\ Criterion(x^*, x'^*)$
7: **until** a time limit t has been reached **or** an optimal solution has been found
8: **return** best solution found by the search algorithm

wards. Appropriate cooling reduces the defects in the resulting crystal. This strategy has been applied successfully to many optimization problems.

The basic principle of the simulated annealing technique implemented in this thesis are described in Algorithm 6 in pseudo code notation. The algorithm creates an initial solution for the considered optimization problem and determines an initial and final cooling temperature, T_{init} and T_{final}, as well as a decay rate α. In each iteration we consider a neighborhood consisting of a single solution. If that solution is better than the current one it will be accepted for the next iteration. Otherwise, the generated neighborhood solution is accepted with a probability depending on the temperature and the quality of the solution. Typically the temperature is very large in the beginning of the search and

Algorithm 4 Hill Climbing(x, H, s)

```
1: repeat
2:    compute N(x, s) a local neighborhood of x of size s
3:    evaluate N(x, s)
4:    select the best feasible solution x' ∈ N(x, s)
5:       if x' < x then
6:          x = x'
7:       end if
8:    end select
9: until solution has not been improved within last H iterations
10: return best solution found by the search algorithm
```

Algorithm 5 Perturbation(x, P)

```
1: repeat
2:    select a random feasible solution x' in a local neighborhood of x
3:       x = x'
4:    end select
5: until P moves worsening the quality of x have been performed
6: return x
```

the simulated annealing behaves like a random search technique. Toward the end of the search the temperature decreases and simulated annealing behaves like an ordinary hill climbing strategy, meaning that it accepts only neighboring solutions of better quality.

Our implementation of a simulated annealing algorithm is controlled by three parameters, the initial temperature T_{init}, the final temperature T_{final} and the decay rate α determining how fast the temperature is lowered. At the start of the algorithm these parameters are determined as follows: First of all we apply a series of moves, to the initial solution and store the fitness loss resulting from each move. Moreover we also record the running times needed to compute and evaluate each move. From the observed fitness losses we compute the average fitness loss per move. T_{init} is chosen such that a move with an average fitness loss has a probability of p_{init} to be performed at the beginning of the simulated annealing algorithm. T_{final} is computed such that a move with an average fitness loss will be performed with probability p_{final} at the end of the simulated annealing strategy. From the recorded running times we estimate roughly how many moves will be computed and evaluated until our algorithm has consumed half of its running time. The value for the decay parameter α is chosen in such a manner that the final temperature will be reached approximately at the middle of the simulated annealing algorithm.

Algorithm 6 Simulated annealing(t, p_{init}, p_{final})

compute an initial solution x
compute T_{init}, T_{final} and α on the basis of p_{init}, p_{final} and *time limit*

$T = T_{init}$
repeat

  select a random feasible solution x' in the local neighborhood of x
    if $x' \leq x$ **then**
      $x = x'$
    else
      $p = e^{-\frac{x'-x}{T}}$
      with probability p **do**
        $x = x'$
      end with
    end if
  end select

  $T = max(T \times \alpha, T_{final})$

until time limit t has reached **or** an optimal solution has been found
return the best solution found by the search algorithm

Chapter 3

A Break Scheduling Problem for Supervisory Personnel

In this chapter, we address a complex real-world break-scheduling problem for supervisory personnel. Supervisory personnel spend most of their workday in front of computer monitors, addressing critical and constantly changing situations. For employees working under such conditions to always maintain high levels of concentration, it is essential that they take occasional breaks. Usually, the amount of break time, as well as the position and duration of breaks within their work time (shift) are regulated by labor rules that must be satisfied by a feasible shift plan. Moreover, to guarantee effective supervision, a minimum number of employees must be working at any given time. In our particular problem, we had to design shift plans over one week containing more than a hundred shifts and more than a thousand breaks. The problem's size and complexity made it impossible for a professional planner to reach a good solution in a reasonable amount of time. Thus, automatic or computer-aided break scheduling was the only way to obtain high-quality shift plans that could both fulfill legal requirements and reduce cost. So, we developed a min-conflicts-based local search algorithm to help planners design such shift plans in the area of supervisory personnel. This heuristic mimics human experts when solving break-scheduling problems and obtains good results. The heuristic is part of a commercial product called Operating Hours Assistant 3.6.

3.1 Formal Inputs to the Break-Scheduling Problem for Supervisory Personnel

In this break-scheduling problem, we are concerned with shift plans for supervisory personnel in which each shift must contain a certain amount of break time. Our goal

is to schedule breaks within the shifts in a solution that minimizes a weighted sum of constraint violations representing legal demands, staffing requirements, and ergonomic criteria. Formally, the break-scheduling problem for supervisory personnel has the following inputs:

- ▷ A planning period is formed by T consecutive time slots $[a_1, a_2), [a_2, a_3), ..., [a_T, a_{T+1}]$, all having the same length (typically 5 minutes). Time points a_1 and a_{T+1} represent the beginning and end of the planning period.
- ▷ Shifts $(s_1, s_2, ..., s_n)$ representing employees working within the planning period. Each shift, s_i, has an adjoined parameter, $s_i.breaktime$, that specifies the required amount of break time for s_i in time slots.
- ▷ The staffing requirements for the planning period are defined as follows. For each time slot, $[a_t, a_{t+1})$, an integer value r_t indicates the required number of employees that should be working during that time slot. An employee is considered to be working during time slot $[a_t, a_{t+1})$ if that employee neither has a break during time slot $[a_t, a_{t+1})$ nor has stopped working at time point at. After a break, an employee needs a full time slot, usually 5 minutes, to become reacquainted with the altered situation. Thus, during the first time slot following a break, an employee is not considered to be working.

Shifts and breaks are characterized by two parameters, start and end, representing the time slots in which a shift or break starts and ends. Subtracting the value for start from the value for end gives the duration of shifts and breaks in time slots. The durations of shifts and breaks are stored in an additional parameter, duration. Moreover, each break is associated with a certain shift in which it is scheduled. We distinguish between two different types of breaks: lunch breaks and monitor breaks.

3.2 Feasible Break-Scheduling Solution

Given a planning period, a set of shifts, the associated total break times, and the staffing requirements, a feasible solution to the break-scheduling problem is a set of breaks with the following characteristics:

- ▷ Each break, b_j, lies entirely within its associated shift, s_i. That is,

$$s_i.start \leq b_j.start \leq b_j.end = s_i.end$$

▷ Two distinct breaks (b_j, b_k) associated with the same shift, s_i, do not overlap in time:

$b_j.start \leq b_j.end \leq b_k.start \leq b_k.end$ or

$b_k.start \leq b_k.end \leq b_j.start \leq b_j.end$

▷ In each shift, s_i, the sum of durations of its associated breaks equals the required amount of break time:

$\sum_{b_j \in s_i} b_j.duration = s_i.breaktime$

3.3 Criteria for Finding an Optimal Solution

Among all feasible solutions for the break-scheduling problem, we try to find an optimal one according to seven criteria, which we model as soft constraints on a solution.

C_1**: Break Positions.** Each break, b_j, may start, at the earliest, a certain number of time slots after the beginning of its associated shift s_i, and may end, at the latest, a given number of time slots before the end of its shift:

$$\begin{array}{lcll} b_j.start & \geq & s_i.start & + \quad distance\ to\ shift\ start \\ b_j.end & \leq & s_i.end & - \quad distance\ to\ shift\ end \end{array}$$

C_2**: Lunch Breaks.** A shift s_i can have several lunch breaks, each required to last a specified number of time slots (*min lunch break duration*), and should be located within a certain time window after the shift start. Let b_{lb} be a lunch break. Then,

$$\begin{array}{lcll} b_{lb}.start & \geq & s_i.start & + \quad distance\ to\ shift\ start\ lb \\ b_{lb}.end & \leq & s_i.end & - \quad distance\ to\ shift\ end\ lb \end{array}$$

C_3**: Duration of Work Periods.** Breaks divide a shift into several work and rest periods. The duration of work periods within a shift must range between a required minimum and maximum duration:

$$\begin{array}{lcllcl} min\ work\ duration & \leq & b_1.start & - \ s_i.start & \leq & max\ work\ duration \\ min\ work\ duration & \leq & b_{j+1}.start & - \ b_j.end & \leq & max\ work\ duration \\ min\ work\ duration & \leq & s_i.end & - \ b_m.end & \leq & max\ work\ duration \end{array}$$

where $(b_1, ..., b_j, b_{j+1}, ..., b_m)$ are the breaks of s_i in temporal order.

C_4**: Minimum Break Times after Work Periods.** If the duration of a work period exceeds a certain limit, the break following that period must last a given minimum number of time slots (min ts count):

$$\begin{array}{lllllllll} b_1.start & - & s_i.start & \geq & work\ limit & \Rightarrow & b_1.duration & \geq & min\ ts\ count \\ b_{j+1}.start & - & b_j.end & \geq & work\ limit & \Rightarrow & b_{j+1}.duration & \geq & min\ ts\ count \end{array}$$

where, once again, $(b_1, ..., b_j, b_{j+1}, ..., b_m)$ are the breaks of s_i in temporal order.

C_5**: Break Durations.** The duration of each break, b_j, must lie within a specified minimum and maximum value:

$$min\ duration \leq b_j.duration \leq max\ duration$$

C_6**: Shortage of Employees.** In each time slot, $[a_t, a_{t+1})$, at least r_t employees should be working.

C_7**: Excess of Employees.** In each time slot, $[a_t, a_{t+1})$, a_t most r_t employees should be working.

3.4 Objective Function

For each constraint, we define a violation degree, $violation(C_k)$, specifying the deviation (in time slots or employees) from the requirements stated by the respective constraint. The importance of each criterion and its corresponding constraint varies from task to task. Consequently, the break-scheduling problem's objective function is the weighted sum of the violation degrees of all the constraints:

$$F(solution) = \sum_{k=1}^{7} W_k \times violation(C_k)$$

where W_k is a weight indicating the importance assigned to constraint C_k. Given an instance of the break-scheduling problem, our goal is to find a feasible solution that minimizes this objective function.

3.5 Solving the Break-Scheduling Problem

Widl [58] has shown that the break scheduling problem for supervisory problem is NP-complete, if all possible break patterns are given explicitly in the input. Local-search techniques represent one possible way to obtain solutions of sufficient quality for

complex optimization tasks. Therefore, we developed a local-search algorithm for the break-scheduling problem, namely a minimum-conflicts-based heuristic [42]. Obtaining a minimum-conflicts-based heuristic for the break-scheduling problem required

- ▷ developing a representation of a solution for the break-scheduling problem,
- ▷ finding a method to generate an initial solution for this problem,
- ▷ defining an objective function to map solutions to real values, and
- ▷ developing moves for the break-scheduling problem to compute the local neighborhood of solutions.

3.6 Solution Representation

We represent a solution of this problem as a set of breaks. For each shift, s_i, the breaks to be scheduled are instantiated at the beginning of a local-search algorithm. So, we first generate lunch breaks, and then we distribute the remaining break time among monitor breaks until the total amount of break time in s_i equals $s_i.breaktime$. Hence, the duration of each lunch break is set to the exact number of time slots required by constraint C_2 (lunch breaks), and the duration of each monitor break lies within the specified minimum and maximum limits imposed by constraint C_5 (break durations). Table 3.1 describes which breaks are created for an 8-hour shift in a problem instance with the settings presented in Figure 3.2.

In the min-conflicts-based algorithm, the start of a break, $b_j.start$, is a variable integer value that can be altered during the search process. In contrast, we require that the duration of a break, $b_j.duration$, remains unchanged and keeps its initially assigned value. We allow that two or more breaks may be scheduled consecutively so that breaks of longer duration can be created.

3.7 Initial Solution and Objective Function

Once the breaks are created, they must be placed in the given shift plan. We implemented two methods to schedule breaks within their associated shifts. The first simply schedules breaks randomly so that they do not overlap. The second schedules breaks so that the resulting break pattern completely satisfies constraints C_1 through C_5. We formulated this task as a simple temporal problem (STP) [16] and we solved this problem by applying a randomized version of the Floyd-Warshall shortest-path algorithm [45] which has a polynomial runtime.

Shift and break information	**Time**	**No. of time slots**
Shift s_i		
$s_i.duration$	8 hrs.	96
$s_i.breaktime$	90 min.	18
Created breaks		
1 lunch break	30 min.	6
6 monitor breaks	10 min. $\times$ 6 = 1 hr.	12

Table 3.1: Example describing which breaks are created for an 8-hour shift with the settings of Figure 3.2.

An STP consists of a set of variables $X = X_1, ..., X_n$ and a set of constraints on those variables. The variables of an STP represent time points having continuous domains. Each constraint is represented as an interval that either restricts the domain values for a single variable X_i or restricts the difference $(X_j - X_i)$ of two distinct variables (X_i, X_j).

To schedule breaks correctly with respect to constraints C_1 through C_5, we modeled the start and end parameters of shifts and breaks as variables of an STP. For the various limits imposed on break positions and on the duration of breaks and work periods, we introduced the following constraints into the STP for constraints C_1 through C_4.

$$C_1: \quad \begin{aligned} b_j.start &\in [(s_i.start + distance\ to\ shift\ start), s_i.end] \\ b_j.end &\in [s_i.start, (s_i.end - distance\ to\ shift\ end)] \end{aligned}$$

$$C_2: \quad \begin{aligned} b_{lb}.start &\in [(s_i.start + distance\ to\ shift\ start\ lb), s_i.end] \\ b_{lb}.end &\in [s_i.start, (s_i.end - distancetoshiftendlb)] \end{aligned}$$

$$C_3: \quad \begin{aligned} b_1.start - s_i.start &\in [min\ work\ duration, max\ work\ duration] \\ b_{j+1}.start - b_j.end &\in [min\ work\ duration, max\ work\ duration] \\ s_i.end - b_m.end &\in [min\ work\ duration, max\ work\ duration] \end{aligned}$$

$$C_4: \quad \begin{aligned} if \quad & b_1.duration \leq min\ ts\ count \\ & b_1.start - s_i.start \in [min\ duration, work\ limit) \end{aligned}$$

$$\begin{aligned} if \quad & b_{j+1}.duration \leq min\ ts\ count \\ & b_{j+1}.start - b_j.end \in [min\ duration, work\ limit) \end{aligned}$$

The two temporal constraints for C_4 are inserted if and only if ($b_1.duration \leq min$

length) and (b_{j+1}.*duration* $\leq$ *minlength*), respectively. Constraint C_5 is automatically satisfied by the obtained solution, because we created only breaks whose durations ranged between the required minimum and maximum break time limits. The criteria for a feasible solution are implicitly covered by the constraints just described. As the objective function, we use the weighted sum of the violation degrees of all constraints, F(*solution*), as discussed earlier.

3.8 Moves and Local Neighborhood

We developed two types of moves for the break-scheduling problem. The first move (called assignment) assigns to a break a new start within its respective shift. The second move (denoted swap) exchanges the start times of two breaks associated with the same shift, meaning those breaks are actually swapped. Figure 3.1 illustrates these two moves. Given a feasible solution S to the break-scheduling problem, the neighborhood $N(S)$ is the set of all solutions obtained by applying an assignment to a single break in S or by swapping two breaks within the same shift in S.

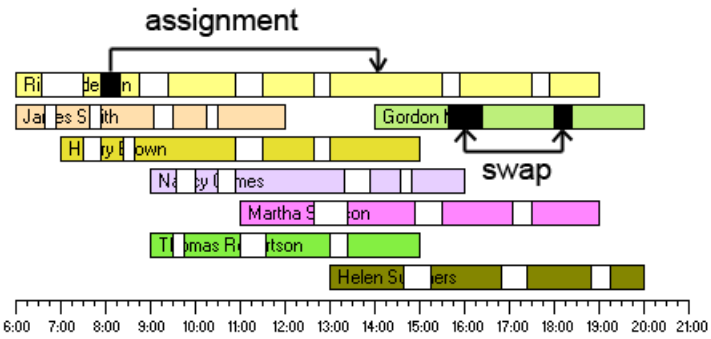


Figure 3.1: The two moves developed for the break-scheduling problem. The assignment move assigns to a break a new start within its respective shift. The swap move exchanges the start times of two breaks associated with the same shift.

3.9 Minimum-Conflicts Heuristics

The minimum-conflicts heuristics tries to improve the current solution by concentrating only on the parts that cause constraint violations. During an iteration, the minimum-conflicts-based heuristic selects a break that violates a constraint and determines a move to minimize, or at least not worsen, the current solution's violation degree. If such a move exists, it is applied to the current solution, and the search continues until some halting condition is satisfied.

The minimum-conflicts search method applies only moves that do not decrease the current solution's quality. Thus, if the search reaches a local-optimum solution that is better than any solution within that solution's neighborhood, the algorithm will not proceed any further, since it will not find solutions of better quality than the local optimum. To avoid this undesirable behavior, we apply an additional strategy named random walk, which has been used successfully in algorithms for satisfiability problems [50].

The random-walk strategy also selects a break that violates a constraint. However, unlike the minimum-conflicts-based heuristic, random walk applies an arbitrary move to that break. On the one hand, the violation degree of the resulting solution could be worse than the previous one. But, on the other hand, performing such moves can help the algorithm escape from local optima. We call the combination of both strategies min-conflicts-random-walk. The random-walk strategy is carried out with a small probability of p, whereas the ordinary minimum-conflicts search is carried out with a high probability of $1 - p$. The concrete value of p is determined experimentally.

3.10 Benchmark Instances for the Break-Scheduling Problem

To evaluate the min-conflicts-random-walk heuristic, we tested it with 20 real-world instances originating from a consulting project. In each instance, we were given the staffing requirements for one week and a shift plan consisting of 120 to 150 shifts, resulting in more than a thousand breaks to be scheduled for each instance.

In addition to the real-world examples, we created a series of randomly generated benchmarks for the break-scheduling problem, which are available at `www.dbai.tuwien.ac.at/proj/SoftNet/Supervision/Benchmarks` and can be used by other researchers to compare their results with ours. (This Web site also details how the random instances are generated.) For random instances, an optimal solution without any constraint violation is known. Thus, we can determine the degree to which the results returned by the min-conflicts-random-walk algorithm deviate from the optimal solution.

3.11 Experimental Settings

In our experiments, the goal of optimization was to obtain solutions that both satisfy labor rules and avoid periods of employee shortages. Labor rules for supervisory personnel are modeled by constraints C_1 (*break positions*), C_3 (*duration of work periods*), and C_4 (*minimum break times after work periods*). So, we assigned the highest weights to these constraints in the objective function. We also gave a high weight to constraint C_6 (*shortage of employees*) and C_2 (*lunch breaks*), and we gave all other constraints lower values. The summary form presented in Figure 3.2 states the exact weights assigned to each constraint.

We conducted our experiments on several computers and normalized all runtimes to a Genuine Intel T2400 laptop running at 1.8 GHz with 2 Gbytes of RAM. For each instance, we performed 10 runs of the min-conflicts-random-walk heuristic, and a single run took 1 hour. In a preliminary series of experiments, we determined the value for the random-walk probability. We considered five different percentages (0, 1, 2.5, 5, and 10) and applied them to our benchmark instances. The best results were obtained with a random-walk probability of $p = 2.5$ percent, so we used this random-walk probability for our further experiments.

3.12 Tests on Real-World Instances

For the 20 real-world benchmarks, we experimented with two variants of initial solutions to assess whether the initial solution affected our algorithm's outcome. The first variant uses an initial solution that randomly places breaks. The second variant begins with a solution that already satisfies constraints C_1 through C_5 by solving the corresponding STP [16].

Table 3.2 presents the results for these benchmarks from 10 runs of the min-conflicts-random-walk algorithm. The mean objective values of the initial solutions obtained by scheduling breaks randomly were six to nine times worse than those of solutions created by solving the corresponding STP. Nevertheless, the min-conflicts-random-walk algorithm can return good solutions for both variants. On the basis of these results, we conclude that the initial solution does not significantly impact the outcome of the solution computed by the min-conflicts-random-walk algorithm.

Total break time per shift (minutes)

For shifts of length *L*	[00:00, 10:00]	minimum break time = floor[(*L* - 20)/50] * 10.
For shifts of length *L*	(10:00, *]	minimum break time = *L*/4.

[*L* denotes shift length (minutes)]

			Weight
Break positions			**20**
Earliest start	00:30	after shift begins.	
Latest end	00:30	after shift ends.	
Lunch Breaks			**10**
Shifts of length	[00:00, 06:00]	do not contain any lunch breaks.	
Shifts of length	(06:00, *]	contain one lunch break.	
Lunch break 1			
Length	00:30	after shift begins.	
Earliest start	03:30	after shift begins.	
Latest end	06:00		
Duration of work periods			**20**
Minimum length	00:30		
Maximum length	01:40		
Minimum break time after work periods			**20**
Schedule a break of minimum	00:20	if work period exceeds 00:50.	
Break durations			**1**
Minimum length	00:10		
Maximum length	01:00		
Shortage of employees			**10**
Excess of employees			**2**

Figure 3.2: Common constraint settings for the considered break-scheduling problem. We used these settings for the experimental evaluation of the min-conflicts-random-walk algorithm.

			min-conflicts-random-walk heuristic with randomly generated initial solutions (objective-function values)				min-conflicts-random-walk heuristic with initial solutions created by solving the corresponding STP (objective-function values)			
Instance	Shifts	Breaks	Initial-solution mean	Best	Mean	Standard deviation	Initial-solution mean	Best	Mean	Standard deviation
2fc04a	135	1,113	95,898.6	3,094	3,248.0	83.6	13,058.8	3,112	3,224.2	86.1
2fc04a03	134	1,130	93,753.8	3,100	3,229.0	60.9	13,092.0	3,138	3,199.6	38.7
2fc04a04	137	1,144	97,175.0	3,232	3,371.2	67.9	12,962.0	3,234	3,342.1	59.5
2fc04b	126	1,064	88,225.6	2,017	2,104.1	91.5	13,639.6	1,822	2,042.8	99.1
3fc04a	124	1,048	90,065.8	1,746	1,809.0	49.1	13,420.0	1,644	1,767.0	101.6
3fc04a03	123	1,052	88,824.6	1,632	1,804.2	87.0	13,202.8	1,670	1,759.2	53.1
3fc04a04	128	1,075	90,801.0	1,942	2,032.0	51.0	12,802.8	1,932	1,980.2	40.4
3si2ji2	151	1,182	98,761.2	3,626	3,692.2	35.2	10,724.4	3,646	3,666.6	14.5
4fc04a	124	1,050	85,808.6	1,694	1,850.6	125.8	13,345.6	1,730	1,817.1	48.2
4fc04a03	123	1,053	89,474.2	1,666	1,795.0	86.8	13,061.2	1,748	1,834.2	55.5
4fc04a04	127	1,068	90,357.4	1,918	2,016.6	94.9	12,808.4	1,982	2,063.6	62.3
4fc04b	125	1,048	89,000.4	1,440	1,526.6	56.3	12,934.4	1,410	1,489.2	48.7
50fc04a	130	1,091	94,776.6	1,750	1,860.6	94.9	14,161.6	1,672	1,827.3	80.6
50fc04a03	130	1,101	92,295.5	1,718	1,847.0	96.3	14,127.3	1,686	1,813.2	84.1
50fc04a04	131	1,112	93,427.4	1,790	1,985.4	83.3	13,930.8	1,792	1,917.2	64.1
50fc04b	126	1,069	91,969.6	1,854	2,012.2	90.9	14,857.6	1,822	1,953.9	77.1
51fc04a	129	1,081	90,374.2	2,048	2,204.2	89.4	14,535.2	2,054	2,166.2	62.3
51fc04a03	129	1,089	92,096.6	2,004	2,096.2	60.4	14,337.2	1,950	2,050.4	86.5
51fc04a04	130	1,101	94,761.6	2,058	2,194.8	64.4	14,319.2	2,116	2,191.4	53.1
51fc04b	126	1,065	89,693.2	2,380	2,513.6	106.2	14,956.4	2,244	2,389.4	93.9

Table 3.2: Test results for 20 real-life benchmark instances from 10 runs of the min-conflicts-random-walk algorithm for randomly generated initial solutions and for solutions created by solving the corresponding simple temporal problem (STP).

3.13 Tests on Random Instances with a Known Optimal Solution

For the experiments with randomly created benchmarks, we selected 10 instances having between 120 and 180 shifts, since that was also the typical instance size of our benchmarks. To these instances, we applied only the min-conflicts-random-walk algorithm using a randomly scheduled initial solution. We created the 10 instances by modeling and solving STPs, so we did not want to create an initial solution using the same technique. Table 3.3 reports the best and mean objective values returned by the min-conflicts-random-walk algorithm in 10 runs and presents the corresponding standard deviations. Given that we assigned high values to single constraints in our objective function, the returned mean and best objective values are quite satisfactory, although an optimum solution avoiding any constraint violations could not be found by the min-conflicts-random-walk algorithm.

Instance	Shifts	Breaks	Optimal	Best	Mean	Standard deviation
random1-1	137	962	0	1,728	1,972.4	176.9
random1-2	164	1,060	0	1,654	1,994.0	172.1
random1-5	141	950	0	1,284	1,477.0	199.0
random1-7	157	1,089	0	1,860	1,077.2	153.9
random1-9	151	985	0	1,358	1,658.0	212.8
random1-13	124	884	0	1,264	1,535.2	245.2
random1-24	137	928	0	1,586	1,712.8	74.5
random1-28	124	809	0	1,710	2,020.0	233.0
random2-1	179	1,255	0	1,686	1,855.2	142.1
random2-4	162	1,075	0	1,712	2,052.8	242.0

Table 3.3: Test results (objective-function values) for 10 benchmark instances with a known optimal solution from 10 runs of the min-conflicts-random-walk algorithm for randomly generated initial solutions.

3.14 Quality of Obtained Solutions

Table 3.4 summarizes the properties of the best solutions obtained by the min-conflicts-based heuristic in our previous experiments. For each instance and constraint, we provide the number of shifts in which the corresponding constraint was violated. We also present the total number of violated shifts per instance, as well as the number of time slots in which employee shortages or excesses occurred.

For the 20 real-world benchmarks, the constraints reflecting legal requirements - C_1 (*break positions*), C_3 (*duration of work periods*), and C_4 (*minimum break times after work periods*) - were completely satisfied in nearly all instances, and the percentage of shifts violating a constraint was less than 5 percent for each instance. In fact, most constraint violations were due to lunch breaks that were not scheduled within their preferred region. In practice, these violations obviously are not considered as serious as violating legal requirements.

Considering the shortage of employees for the real-world examples, we observe that in most instances shortages occurred in less than 5 percent of the entire planning period. Only for instances 50fc04b, 51fc04a, and 51fc04b was the min-conflicts-based heuristic unable to compute a solution under a 5 percent shortage threshold. Also, the high excess of employees reported for some instances was due to the characteristics of the given shift plan for that problem; hence, high excess percentages were unavoidable for those instances.

Regarding the best solutions for the 10 randomly created instances, constraints reflecting legal requirements were completely satisfied in three instances (random1-1, random1-5, and random 1-24). Moreover, for six other instances, those constraints were violated only in a few shifts. Only for instance random1-9 did the obtained best solution have seven shifts violating constraint C_3 (*duration of work periods*) or C_4 (*minimum*

		No. of shifts violating constraint					Shifts with violations		Time slots with			
				Duration	Minimum				Shortage		Excess	
		Break	Lunch	of work	break	Break						
Instance	Shifts	positions	breaks	periods	times	durations	No.	%	No.	%	No.	%
2fc04a	135	0	2	0	0	0	2	1.5	55	2.7	658	32.3
2fc04a03	134	0	3	0	0	0	3	2.2	55	2.7	684	33.6
2fc04a04	137	0	3	0	0	0	3	2.6	53	2.2	726	35.5
2fc04b	126	0	0	0	0	1	1	0.8	72	3.5	379	18.6
3fc04a	124	0	3	0	0	0	3	2.4	37	1.8	410	20.1
3fc04a03	123	0	1	0	0	0	1	0.8	27	1.3	463	22.7
3fc04a04	128	0	0	0	0	0	0	0	27	1.3	524	25.7
3si2ji2	151	0	0	0	0	0	0	0	2	0.1	1,093	53.7
4fc04a	124	0	3	0	0	0	3	2.4	41	2.0	412	20.2
4fc04a03	123	0	0	0	0	0	0	0	31	1.5	445	21.8
4fc04a04	127	0	2	0	0	0	2	1.6	28	1.4	538	26.3
4fc04b	125	0	0	0	0	0	0	0	37	1.8	357	17.5
50fc04a	130	0	2	0	0	0	2	1.5	74	3.6	284	13.9
50fc04a03	130	0	3	0	0	0	3	2.3	55	2.7	343	16.8
50fc04a04	131	0	5	0	0	0	5	3.8	50	2.5	402	19.7
50fc04b	126	0	1	1	0	0	1	0.8	117	5.7	196	9.6
51fc04a 1	29	0	2	0	0	0	2	1.6	108	5.3	263	12.9
51fc04a03	129	0	2	0	0	1	3	2.3	85	4.2	309	15.2
51fc04a04	130	0	3	0	0	0	3	2.3	85	4.2	363	17.8
51fc04b	126	0	2	1	0	0	3	2.4	137	6.7	198	9.7
random1-1	137	0	0	0	0	0	0	0	38	2.0	114	6.0
random1-2	164	0	0	0	3	0	3	1.8	89	4.4	196	9.7
random1-5	141	0	0	0	0	0	0	0	89	4.6	182	9.4
random1-7	157	0	0	0	1	0	1	0.6	43	2.2	142	7.1
random1-9	151	0	2	6	1	0	9	6	67	3.3	162	8.0
random1-13	124	0	0	1	1	0	2	1.6	73	3.6	162	8.0
random1-24	137	0	1	0	0	0	1	0.7	106	5.3	178	8.9
random1-28	124	0	0	0	1	0	1	0.8	100	5.0	169	8.4
random2-1	179	0	0	2	0	0	2	1.1	100	5.0	238	11.8
random2-4	162	0	1	1	1	0	3	1.9	93	4.6	174	8.6

Table 3.4: Constraint violations of the best solutions obtained by the min-conflicts-based heuristic for real-life and randomly generated benchmark instances.

break times after work periods). In nine instances, the percentage of time slots with a shortage of employees did not exceed 5 percent, and in random1-24 the percentage of understaffed time slots (5.3 percent) was very close to this threshold.

Finally, Figure 3.3 presents part of the best solution obtained for instance 2fc04a04 and shows the curve of required, present, and working employees for that time period. The minimum staffing requirements were violated only once, for 5 minutes, during this time period. Other than two lunch breaks not in their preferred time ranges, all constraints, including those modeling labor rules, were satisfied completely.

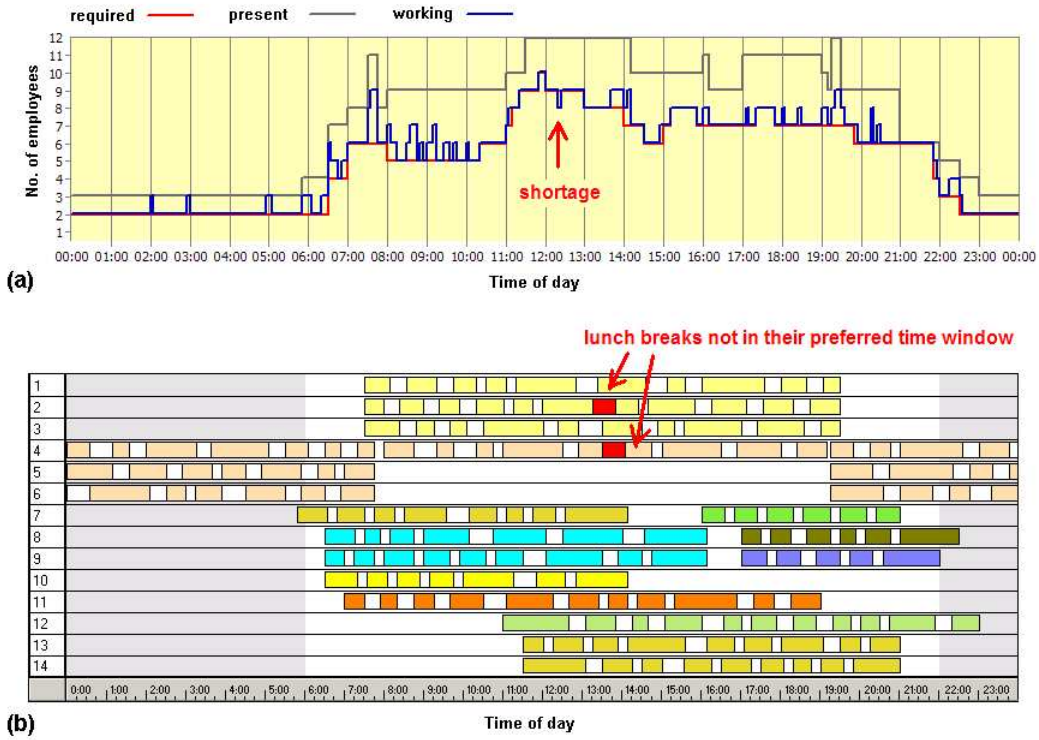


Figure 3.3: Part of the best solution found for instance 2fc04a04: (a) the number of required, present, and working employees, and (b) the shift plan for this same time period. All constraints were satisfied completely except for two lunch breaks that were not in their preferred time ranges.

Chapter 4

Scheduling Breaks in Shift Plans of Call Centers

In this section we consider a further real-life break scheduling problem originating from a call center. Although the addressed problem has similar characteristics as the break scheduling problem for supervisory personnel from Chapter 3 there are significant differences in the constraints involved in the problem.

4.1 Problem Description

Formally, as input for the call center break scheduling problem we are given:

- ▷ a *planning period* formed by T consecutive time slots $[a_1, a_2), [a_2, a_3), ..., [a_T, a_{T+1})$ all having the same length *slotlength* (in minutes). Time points a_1 and a_{T+1} represent the beginning and end of the planning period. All time points have the same format *day:hour:minute*.
- ▷ *n shifts* $(s_1, s_2, ..., s_n)$ representing employees working within the planning period. Each shift s_i has the adjoined parameters, $s_i.start$ and $s_i.duration$ representing its start and its duration. Each shift corresponds to exactly one employee.
- ▷ *break quantities* and *break types* to be scheduled for each shift. We distinguish between two different types of breaks: lunch breaks and monitor breaks. The parameter $s_i.lunch$ stores the duration of a shift's lunch break in minutes, whereas the parameter $s_i.monitor$ specifies a shift's monitor break quantity.

- ▷ the *staffing requirements* for the planning period. Each time slot $[a_t, a_{t+1})$ has an adjoined integer value r_t indicating the optimal number of employees that should be working during time slot $[a_t, a_{t+1})$. An employee is considered to be *working* during time slot $[a_t, a_{t+1})$ if in its corresponding shift no break is scheduled during time slot $[a_t, a_{t+1})$.

A *break* b is characterized by the parameters, $b.shift$ specifying its associated shift, its start $b.start$ and its duration $b.duration$. We assume that all parameters representing time points coincide with a time point a_t defining the start or the end of a time slot of the planning period. Moreover we expect each parameter representing a duration or a break quantity to be a multiple of *slotlength*.

Given a planning period, a set of shifts, the associated break quantities, and the staffing requirements, a *feasible solution* to the break scheduling problem is a set of breaks such that:

1. Each break lies entirely within its associated shift.
2. Two distinct breaks associated with the same shift do not overlap in time.
3. For each shift the sum of all its associated break durations is exactly the specified break quantity for the shift, that is $s_i.lunch + s_i.monitor = \sum_{b_j \in s_i} d_j.duration$.
4. If $s_i.lunch > 0$ then there is one break in s_i whose duration is at least $s_i.lunch$.

Among all feasible solutions for the break scheduling problem we aim at finding an optimal one according to various criteria. These criteria are modeled as constraints on feasible solutions. Basically we distinguish between four main groups of constraints, namely constraints on:

1. The position of breaks within shifts.
2. The duration of breaks.
3. The distances between breaks.
4. The excesses and shortages of working employees according to staffing requirements.

4.1.1 Constraints on the Position of Breaks within Shifts

C_1 : ***MinimumDistanceToShiftBegin:*** Each break may start not earlier than a given number of minutes after the beginning of its associated shift.

C_2 : ***MinimumDistanceToShiftEnd:*** Each break must end not later than a given number of minutes before the end of its associated shift.

C_3 : ***MaximumDistanceToShiftBegin:*** The earliest break of a shift must not start later than a given number of minutes after the beginning of the shift.

C_4 : ***MaximumDistanceToShiftEnd:*** The latest break of a shift must not end earlier than a given number of minutes before the end of the shift.

4.1.2 Constraints on the Distances Between Breaks

C_5 : ***MinimumDistanceBetweenBreaks:*** The temporal distance between two consecutive breaks must be at least a given minimum number of minutes.

C_6 : ***MaximumDistanceBetweenBreaks:*** The temporal distance between two consecutive breaks must not exceed a given maximum number of minutes.

4.1.3 Constraints on the Duration of Breaks

C_7 : ***MinimumBreakDuration:*** The duration of each break must be at least a given minimum number of minutes.

C_8 : ***MaximumBreakDuration:*** The duration of each break must not exceed a given maximum number of minutes.

C_9. ***OptimumBreakDuration:*** The duration of each break should be equal to a given optimum number of minutes.

C_{10} : ***MinimumDurationAfterDistance:*** If the distance between two consecutive breaks reaches or exceeds a certain number of minutes the duration of the latter break must be at least of a given minimum duration.

4.1.4 Constraints on the Excess and Shortage of Working Employees

C_{11} : ***NoExcess:*** In each time slot $[a_t, a_{t+1})$ the number of working employees, i.e., the employees who are not assigned a break in that time slot, should not exceed r_t.

$C_{12:}$ ***NoShortage:*** In each time slot $[a_t, a_{t+1})$ the number of working employees should be at least r_t.

C_{SD} : ***NoSquaredDeviation:*** In many practical instances for the break scheduling problem the staffing requirements are significantly higher or lower than the number of scheduled employees during the overall planning period. Consequently, each solution will always produce the same amount of excess or shortage. For such instances we introduced an additional constraint aimed at minimizing the squared deviation from staffing requirements in each time slot. Informally speaking, this constraint prefers solutions whose curve of working employees has a shape similar to the curve representing the staffing requirements.

4.1.5 Extending the Problem with Breaks of Fixed Duration

When scheduling breaks within shift plans it is sometimes necessary to constrain a single break differently from the remaining breaks within its shift. For instance, employees prefer to have a one hour lunch break at the middle of their duty or between 11:00 and 14:00. Therefore we introduce a constraint defined on a single break within a shift.

C_{13} : ***FixedBreak:*** Each shift can contain a break of a certain specified duration, which may differ from the durations required by other constraints. Optionally, this break must lie within some given *allowed time range*, preferably within a given *optimum time range*. The break must not be scheduled within a given *forbidden time range*.

Note: The criteria required by the constraint *FixedBreak* may contradict the requirements of several previously introduced constraints. For that reason, the following constraints are not applied to that single break of desired length: *MinimumDistanceToShiftBegin* (C_1), *MinimumDistanceToShiftEnd* (C_2), *MinimumBreakDuration* (C_7), *MaximumBreakDuration* (C_8), and *OptimumBreakDuration* (C_9).

4.1.6 Extending the Problem with Meetings

Call center employees can take part in meetings during their working time. While attending meetings call center agents do not process incoming phone calls. Thus, during the time a meeting takes place the participating employees are not considered to be working with respect to staffing requirements.

Example 1. In the shift plan given in Figure 4.1 call center employees represented by shifts s_2, s_3, and s_4 take part in meeting m_1 taking place from 12:30 until 13:30. Employees working in shifts s_1 and s_4 attend meeting m_2 from 17:30 to 18:30.

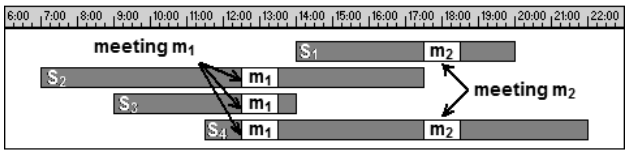


Figure 4.1: A shift plan containing meetings.

In order to handle meetings we have to extend the break scheduling problem further. Moreover we introduce an additional constraint concerning the break time scheduled in meetings. In addition to the input for our basic problem we are given:

- ▷ k meetings $m_1, m_2, ..., m_k$. Each meeting m_j has two adjoined parameters $m_j.start$ and $m_j.duration$, specifying its start time and its duration. Moreover, each meeting has an adjoined set $m_j.S$. Set $m_j.S$ contains shifts and indicates that employees assigned to these shifts participate in meeting m_j. Additionally we are given an integer value q_j specifying the break time required to be scheduled during meeting m_j.

C_{14} : ***BreakQuantityInMeeting:*** For each employee participating in a meeting m_j we require that exactly q_j minutes of break time are scheduled within meeting m_j.

Meetings have the following side effects on several constraints of the basic break scheduling problem:

C_5, C_6, C_{10}**:** These constraints are only relevant for breaks not scheduled during the same meeting. In other words, these constraints are ignored for consecutive breaks ending and starting during the same meeting.

C_7, C_8, C_9**:** We consider only those parts of breaks which are scheduled outside the time range of a meeting, disregard breaks of a certain fixed duration, and refer to these breaks as *breaks outside a meeting*. The constraints on the minimum, maximum, and optimum break duration are only applied to these breaks outside a meeting.

C_{11}, C_{12}, C_{SD}**:** While participating in meetings employees are not considered to be working. Breaks scheduled during meetings do not further decrease the number of working employees.

4.2 Adapting the Min-Conflicts-Based Heuristic for the Call Center Break Scheduling Problem

To solve the call center break scheduling problem we adapted and modified the min-conflicts based heuristic from Section 3.9 as follows:

- ▷ We extended our solution representation with meetings, monitor breaks, lunch breaks and fixed breaks.
- ▷ We modified existing constraints, implemented new constraints, and formulated a new objective function for the call center break scheduling problem.
- ▷ We adapted our definition of feasible moves in accordance with the changed problem structure.

4.2.1 Representation of Solutions for the Break Scheduling Problem

We represent the solution for the call center break scheduling problem as a set of breaks. Each break has a variable start and constant duration. Moreover, each break is associated with a certain shift and must lie entirely within that shift's range. Given a shift and its quantities of lunch and monitor breaks we distribute the break time among the following three types of breaks:

fixed breaks: For each break required by constraint *FixedBreak* (C_{13}) we generate a so-called fixed break having the desired duration. If possible the entire lunch break quantity is part of a single fixed break.

lunch break: If it is not possible to schedule the lunch break within a fixed break we generate a lunch break. Each shift may contain at most one lunch break comprising its total lunch time quantity.

monitor breaks: The remaining time not planned as fixed breaks and lunch breaks is scheduled within monitor breaks. We try to assign a monitor break the optimal break duration as required by constraint *OptimalBreakDuration* (C_9) to each monitor break but the last monitor break may be shorter than the desired optimum duration.

The obtained breaks are scheduled randomly in their respective shifts such that the the obtained solution is feasible and satisfies the constraints $MinimumDistanceToShiftBegin(C_1)$ and $MinimumDistanceToShiftEnd(C_2)$. This solution acts as the initial solution for our proposed local search techniques.

4.2.2 Objective Function

The break scheduling problem can be modeled as a multi-criteria optimization problem where an objective function is to be minimized. The importance of a single criterion and the corresponding constraint varies from task to task. Thus, the break scheduling problem's objective function can be modeled as a weighted sum of the violation degree of each constraint, or more formally:

$$F(Solution) = \sum_{i=1}^{14} W_i \cdot violations(C_i) + \frac{violations(C_{SD})}{2 \cdot ub(C_{SD})}$$

In the objective function presented above, $ub(C_{SD})$ denotes an upper bound on the violation degree of the constraint *NoSquaredDeviation*. If two solutions have the same objective value according to constraints $C_1, ..., C_{14}$ the objective function prefers the solution with a smaller squared deviation from staffing requirements.

4.2.3 Moves and Local Neighborhood

Given a feasible solution S to the break scheduling problem we define its neighborhood $N(S)$ to be the set of all solutions obtained by applying an assignment on a single break in S or by swapping two breaks within the same shift in S as described in Section 3.8. As a *legal move* we consider any assignment or swap guaranteeing that after the move:

1. The breaks in the affected shifts are not overlapping.
2. The lunch break is not scheduled in a meeting.
3. The affected breaks lie within their allowed time regions specified by the constraints *MinimumDistanceToShiftBegin* (C_1), *MinimumDistanceToShiftEnd* (C_2) and *FixedBreak* (C_{13}).
4. Fixed breaks are not preceded or succeeded by any other break after the move.

4.3 Computational Results

4.3.1 Randomly Generated Instances

To assess the quality of solutions returned by the min-conflicts-random-walk heuristic, we wanted to generate instances for which we know that almost all constraints can be satisfied completely. To this aim we built a generator which first builds a solution and derives an instance from it afterwards.

For building a solution we consider the randomly created benchmark instances for the min-shift-design problem ([43]), which can be found under `http://www.dbai.tuwien.ac.at/proj/Rota/benchmarks.html`. From these instances we obtain a shift plan by extracting the shifts in the provided sample solutions starting on the first day.

For each shift we generate a certain number of breaks using the settings of the real-life examples. Breaks are scheduled within their shifts such that all constraints except C_9 *OptimumBreakDuration* are satisfied. The optimum break duration can be violated by some breaks, since the total amount of break time need not be a multiple of the given optimum break duration. The problem of scheduling breaks correctly in a shift is formulated as a simple temporal problem (STP) ([16]) and is solved by applying Floyd-Warshall's all-pairs-shortest algorithm ([45]).

After scheduling breaks we randomly insert meetings in the shift plan. For that purpose we create a meeting lasting 30, 40, 50 or 60 minutes and place it randomly in the planning period. Then we determine the shifts which may contain that meeting and assign the meeting to a random number (two at least) of these shifts . The total number of break time which has to be scheduled in each meeting is set to the amount of break time scheduled in meetings in the current solution.

Finally, for each time slot we set the staffing requirements to the number of working employees (available workers minus workers in meetings or breaks). So we may guarantee that the staffing requirements may be satisfied. The generated instances have a solution satisfying all constraints except the constraint C_9 *OptimumBreakDuration*. A detailed description of the instance generator and the created benchmark instances are available at `http://www.dbai.tuwien.ac.at/proj/SoftNet/Benchmarks/`.

We applied the min-conflicts-random-walk algorithm on 44 randomly generated instances. For each instance we performed ten runs on a Genuine Intel T2400 laptop running at 1.8 GHz with 2 Gbytes of RAM. A single run was executed with a ten minute runtime limit. Table 4.2 provides for each instance the violation degree of the best known solution and the violation degree of the best solution returned in ten runs by the min-conflicts based heuristic. In addition Table 4.2 presents the violation degree of each single constraint for the best solution returned per instance. Note that these violation degrees are not multiplied by their respective weights. To obtain the value in column 'Best Solution Found' those violations must be multiplied by their respective weight.

Considering excess and shortage of working employees the obtained solutions deviate from the known optimal one only by a few percent.

Regarding the various constraints on break positions, distances, durations, fixed breaks and meetings we observe that only the constraints C_5 *MinimumDistanceBetweenBreaks*, C_8 *MaximumBreakDuration* and C_9 *OptimumBreakDuration* are violated. This

	Constraint	Weight W_i
C_1	*MinimumDistanceToShiftBegin*	10
C_2	*MinimumDistanceToShiftEnd*	10
C_3	*MaximumDistanceToShiftBegin*	100
C_4	*MaximumDistanceToShiftEnd*	100
C_5	*MinimumDistanceBetweenBreaks*	10
C_6	*MaximumDistanceBetweenBreaks*	100
C_7	*MinimumBreakDuration*	3
C_8	*MaximumBreakDuration*	3
C_9	*OptimumBreakDuration*	3
C_{10}	*MinimumDurationAfterDistance*	100
C_{11}	*NoExcess*	20
C_{12}	*NoShortage*	20
C_{13}	*FixedBreak*	10
C_{14}	*BreakQuantityInMeeting*	60

Table 4.1: Weights of constraints for the considered real-life instances.

is acceptable because we considered these constraints to be less important and consequently we have assigned smaller weights to them in our evaluation function (see Table 4.1). We also know that constraint C_9 *OptimumBreakDuration* can not be satisfied completely even within an optimal solution.

Any other constraint, in particular the very crucial constraints C_3 *MaximumDistanceToShiftBegin*, C_4 *MaximumDistanceToShiftEnd*, C_6 *MaximumDistanceBetweenBreaks*, C_{10} *MinimumDurationAfterDistance* and C_{14} *BreakQuantityInMeeting* are satisfied completely for each instance. For the reasons just mentioned we conclude that our min-conflicts based heuristic returns solutions of acceptable quality for each of the regarded benchmark instances.

4.3.2 Real-Life Application

The min-conflicts-based algorithm has been applied successfully at a call center, where it is used to compute daily break schedules within a few seconds. At this point we present one solution for a real-life instance. In Figure 4.2 we see the curve of required employees (solid curve) and the curve of working employees (dashed curve) resulting from the best solution found for that problem. The required minimum number of employees is not violated at any time. Table 4.3 presents the objective function value of each constraint for the best solution for the considered instance. We observe that nearly all constraints are completely satisfied. Only the optimum break duration has been violated for some breaks and there exists some excess of working employees, which cannot be avoided due to the characteristics of the considered real-life instance. Moreover, for that particular

Ex.	Number of		Cost of		Timeslots with				Violation Degree of			
									Minimum Distance Between	Maximum Break Duration	Optimum Break Duration	Any Other Constraint
	Shifts	Breaks	Best Known Solution	Best Solution Found	Excess	%	Shortage	%	Breaks			
1	24	95	33	135	2	1.4	2	1.4	1	-	15	-
2	13	47	30	50	-	-	-	-	2	-	10	-
3	9	30	48	48	-	-	-	-	-	-	16	-
4	29	102	72	222	-	-	-	-	15	-	24	-
5	17	63	0	62	-	-	-	-	5	-	4	-
6	39	141	36	258	2	1.3	2	1.3	10	-	26	-
7	31	109	21	123	1	0.7	1	0.7	5	-	11	-
8	29	108	45	195	2	1.3	2	1.3	4	-	25	-
9	15	51	15	15	-	-	-	-	-	-	5	-
10	24	87	24	44	-	-	-	-	2	-	8	-
11	9	28	30	30	-	-	-	-	-	-	10	-
12	24	95	33	123	1	0.7	1	0.7	5	-	11	-
13	13	52	18	48	-	-	-	-	3	-	6	-
14	9	35	42	102	-	-	-	-	6	-	14	-
15	29	114	78	506	2	0.7	2	0.7	33	-	32	-
16	17	63	15	95	1	0.7	1	0.7	4	-	5	-
17	39	147	63	325	1	0.6	1	0.6	18	1	34	-
18	31	109	66	345	4	2.7	4	2.7	11	-	25	-
19	29	108	75	215	2	1.3	2	1.3	6	-	25	-
20	15	56	30	89	1	0.7	1	0.7	1	-	13	-
21	24	95	27	77	-	-	-	-	5	-	9	-
22	9	33	30	30	-	-	-	-	-	-	10	-
23	46	170	51	421	4	2.6	4	2.6	15	-	37	-
24	49	192	84	422	1	0.6	1	0.6	25	-	44	-
25	52	184	36	404	2	1.3	2	1.3	21	-	38	-
26	53	185	144	1130	13	4.1	13	4.1	37	-	80	-
27	50	170	48	334	4	2.7	4	2.7	9	-	28	-
28	45	163	48	318	3	2	3	2	12	-	26	-
29	45	163	30	378	6	3.7	6	3.7	3	-	36	-
30	60	211	156	1332	18	5.8	18	5.8	36	-	84	-
31	38	136	72	556	5	1.6	5	1.6	26	-	32	-
32	52	182	33	351	1	0.7	1	0.7	23	-	27	-
33	65	240	48	582	6	3.7	6	3.7	21	-	44	-
34	46	170	93	348	2	1.3	2	1.3	13	-	46	-
35	49	196	93	459	3	1.9	3	1.9	18	1	52	-
36	52	184	105	373	2	1.3	2	1.3	17	-	41	-
37	53	206	162	1180	10	3.2	10	3.2	54	-	80	-
38	50	181	102	420	2	1.3	2	1.3	19	1	49	-
39	45	172	63	265	2	1.3	2	1.3	11	-	25	-
40	45	163	84	422	5	3.1	5	3.1	9	1	43	-
41	60	234	192	1612	16	5.2	16	5.2	66	-	104	-
42	38	146	126	738	6	1.9	6	1.9	33	-	56	-
43	52	182	111	467	3	2	3	2	20	-	49	-
44	65	240	108	620	3	1.9	3	1.9	23	3	87	-

Table 4.2: Best solutions obtained for 44 randomly created instances with known optimum solution.

instance the break durations may not be further improved. Consequently, also for that real-life benchmark instance the quality of the computed solution is almost optimal.

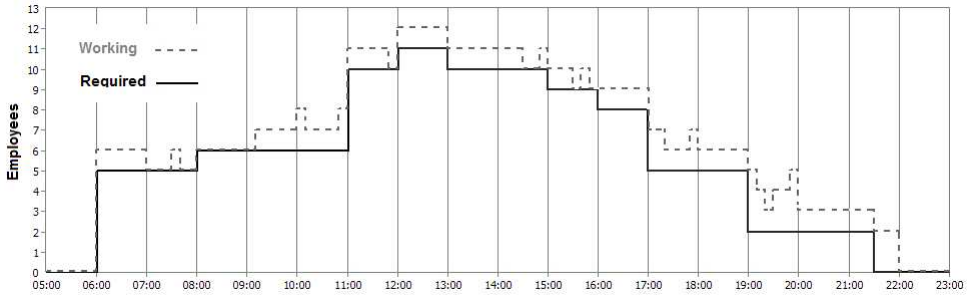


Figure 4.2: Curve of required and working employees resulting from the best solution for a real-life instance of the call center break scheduling problem.

Constraint		Viol. Deg.	Weight	Product
C_1	MinimumDistanceToShiftBegin	0	10	0
C_2	MinimumDistanceToShiftEnd	0	10	0
C_3	MaximumDistanceToShiftBegin	0	100	0
C_4	MaximumDistanceToShiftEnd	0	100	0
C_5	MinimumDistanceBetweenBreaks	0	10	0
C_6	MaximumDistanceBetweenBreaks	0	100	0
C_7	MinimumBreakDuration	0	3	0
C_8	MaximumBreakDuration	0	3	0
C_9	OptimumBreakDuratioin	12	3	36
C_{10}	MinimumDurationAfterDistance	0	100	0
C_{11}	NoExcess	94	20	1880
C_{12}	NoShortage	0	20	0
C_{SD}	NoSquaredDeviation	128	1/9576	0.01
	Objective function value	**1916.01**		

Table 4.3: Detailed results for a real-life instance of the call center break scheduling problem.

Chapter 5

TEMPLE - A Domain Specific Language for Staff Scheduling Problems

In this chapter we design the domain specific language TEMPLE in order to reduce the effort for developing solutions for staff scheduling problems. The name TEMPLE was inspired by Figure 5.1 reflecting our desire to model complex staff scheduling problems in the same modular manner as ancient temples are built-up by many single building blocks. With TEMPLE we want to model staff scheduling tasks in an easy and natural manner, and we would like to solve them via local search algorithms. For that purpose TEMPLE must satisfy the following two demands:

1. TEMPLE must offer abstractions and notations reflecting common features of resource planning and staff scheduling problems. These abstractions and notations must support a user in creating accurate problem models in short time.

2. TEMPLE must provide abstractions and notations corresponding to essential building blocks of local search techniques. These essential building blocks must be sufficient to obtain a *generic local search algorithm* for a particular resource planning and staff scheduling problem. Any further knowledge or information on local search techniques beyond those key building blocks must be masked from an end-user.

5.1 Design Goals for TEMPLE

Modularity In TEMPLE, a problem instance is modeled by small, concise building blocks reflecting common features of staff scheduling problems (Figure 5.1 (a)). New building blocks are derived from already existing ones. By this principle a user is forced to formulate a complex problem in small, concise and traceable steps. Consequently, the resulting problem models are well-structured, easy to understand, modify and maintain.

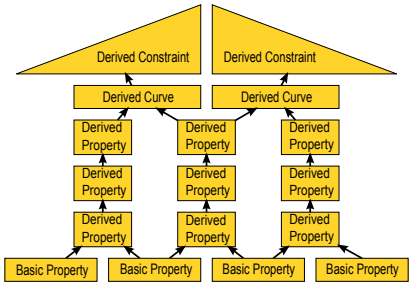
Adaptability and Extensibility Problems modeled in TEMPLE can be adapted easily. A few small changes in a problem's formulation may result only in a few small changes in the model written in the domain specific language (Figure 5.1 (b)). Building blocks that are not affected by changes must remain unchanged. Additional requirements shall be able to be added easily without interfering with other building blocks (Figure 5.1 (c)).

Simplicity TEMPLE demands only basic programming skills from end-users. Anybody familiar with a third generation programming language should be able to understand and use TEMPLE. Concepts of advanced programming paradigms, e.g., object orientation, or knowledge on local search techniques, are not required from a user.

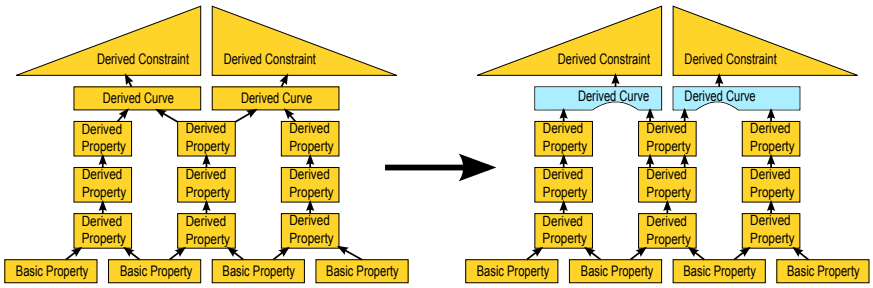
Openness In contrast to other constraint-based modeling languages, TEMPLE is not restricted to a finite set of predefined features or constraints. With TEMPLE arbitrary features or constraints of staff scheduling problems can be modeled.

Automatic Optimization Once a problem is modeled in TEMPLE it can be optimized immediately without requiring additional coding from the user.

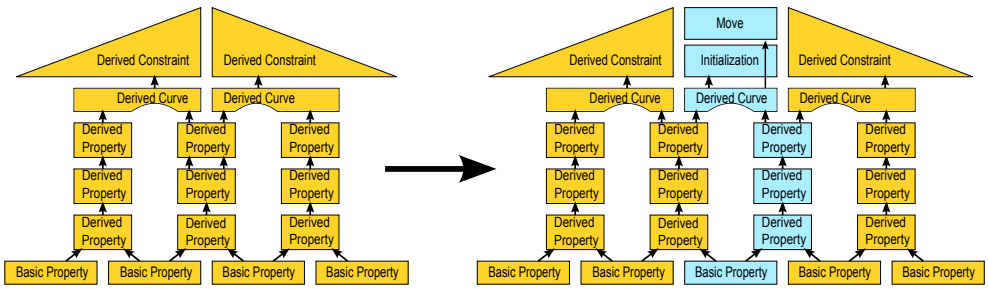
Efficiency TEMPLE represents an additional layer atop a general purpose programming language, providing abstractions and notations focused on a staff scheduling problems. Consequently, staff scheduling tasks can be modeled more easily, concisely and quickly. The drawback of TEMPLE is a certain computational overhead that could be avoided at a lower level of implementation. Thus an important design goal for TEMPLE is that its intrinsic computational overhead is kept as little as possible. Thereby, we ensure that problems are not only modeled effectively but also solved efficiently.



(a) Modularity and Derivation - A problem model is built up by small and simple building blocks. From basic building blocks a user can derive further properties, curves and finally constraints.



(b) Adaptability - A few small changes in the problem formulation result in a few small changes in the problem model. Building blocks not affected by changes are not altered at all.



(c) Extensibility - Additional requirements can be added easily without interfering with previously defined building blocks.

Figure 5.1: Selected design goals which are achieved by the TEMPLE modeling language.

5.2 Building Blocks of Staff Scheduling Problems

5.2.1 Intervals and Links between Intervals

Intervals are central building blocks of staff scheduling problems. Figure 5.2 shows the intervals occurring in the call center break scheduling problem from Section 4. Shifts, breaks, meetings, the time slots of a planning period, and even the entire problem itself can be considered as intervals. As shown in 5.3, every interval is characterized by three basic properties: `Start`, `Duration`, and `End`.

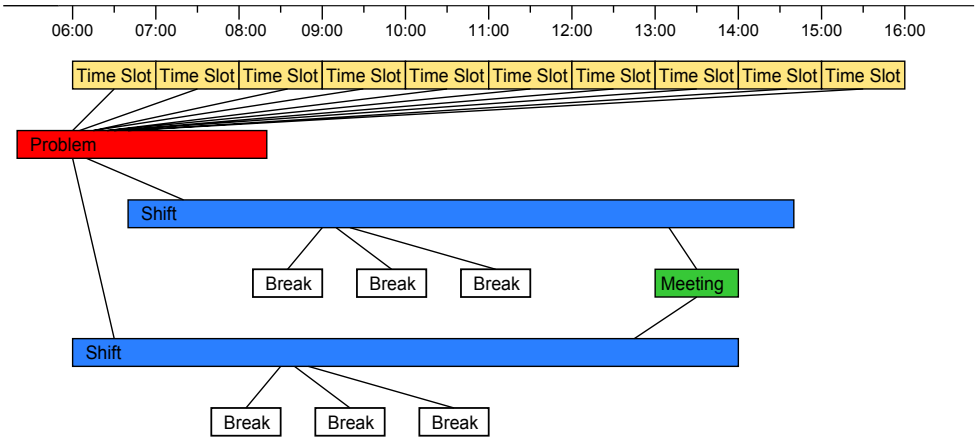


Figure 5.2: The different kinds of intervals and links between intervals involved in the call center break scheduling problem from Section 4.

In addition, we observe that in staff scheduling problems intervals are linked with each other. Figure 5.2 depicts the links between the intervals occurring in the call center break scheduling problem from Section 4. Breaks are linked to the shifts in which they are scheduled, there is a link between a meeting and the shifts representing the employees participating in that meeting, and the entire problem is linked to all shifts as well as the time slots forming the planning period.

Design Decision. In TEMPLE it must be possible to declare different kinds of *intervals* and *links between intervals*. Each kind of interval must have three basic properties, `Start`, `Duration` and `End`.

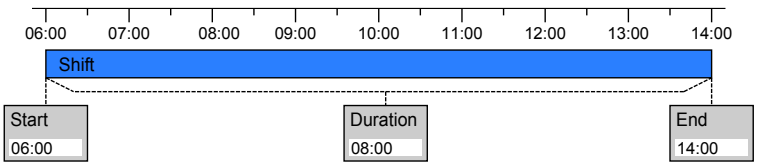


Figure 5.3: A time interval is characterized by three basic properties, `Start`, `Duration` and `End`.

5.2.2 Derived Properties and Constraints

A characteristic of staff scheduling problems is that their features and constraints can be derived step by step one after the other. For instance, in many real-life applications it is common to require that a minimum percentage of break time, e.g., 20%, must be scheduled in each shift. Figure 5.4 shows how the violation degree of that constraint can be computed for a single shift in several steps:

1. We compute the break time scheduled in the shift. For that purpose we consider the two breaks linked with the shift, and sum up the values for their basic property `Duration`. In that way we derive a new property of the shift called `TotalBreakTime` .

2. We compute the shift's break time percentage, by dividing a shift's property `TotalBreakTime` by its `Duration`. Again, we derive a new property of the shift called `TotalBreakTimeInPercent`.

3. We impose the constraint, requiring that a shift's break time percentage must be at least 20%. The violation degree of that constraint is computed by checking a shift's property `TotalBreakTimeInPercent`. The constraint, derived in that manner, is called `MinimumBreakTime` and is associated to the shift.

Design Decision. In TEMPLE it must be possible to *derive new properties and constraints* step by step one after the other. Property values or constraint violation degrees must be computed from already existing properties of an interval or its linked intervals.

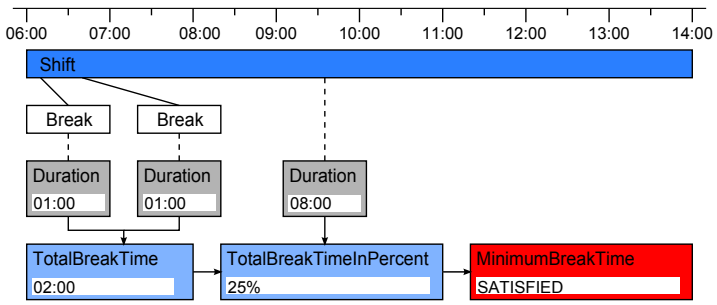


Figure 5.4: In staff scheduling problems properties and constraints are derived step by step from already existing properties.

5.2.3 Derived Curves

Curves represent further central building blocks of staff scheduling problems, which can be used to model many features of staff scheduling tasks such as staffing requirements or available staff. Formally, in the context of this thesis, a curve is a function, mapping each time slot of a considered planning period to a specific value. A curve is derived from intervals, by incrementing or decrementing the curve's values over the duration of single intervals.

For instance, Figure 5.5 presents how the time periods during which an employee is actually working and not having a break can be represented as a curve over time. The curve is incremented over the duration of the shift and decremented along the duration of breaks. Moreover, curves can also be derived from other, already existing curves, e.g., by subtracting staffing requirements from available staff we obtain a curve representing the deviations from staffing requirements.

Design Decision. TEMPLE must provide *derived curves* as further basic building blocks. Curves are derived from basic properties associated to intervals or from other curves. Moreover, it should also be possible to derive properties or constraints from already existing curves associated with an interval.

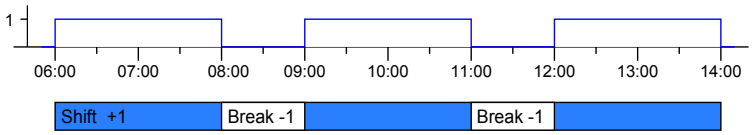


Figure 5.5: A curve modeling the periods while an employee is actually working and not having a break.

5.2.4 Building Blocks of Local Search Techniques

To identify the basic building blocks of local search techniques we reconsider the basic steps within a local search algorithm :

1. We compute an *initial solution* for a specific problem instance.
2. As long as a certain termination condition is not fulfilled we perform the following three steps.
   (a) We compute a set of small changes, also denoted as *moves*, to obtain a local neighborhood of the current solution.
   (b) We evaluate the effect of each move on the current solution. When evaluating a move we determine the change within the problem's *objective function* resulting from the move.
   (c) We select a move and apply it to obtain a new solution. Usually, a move is selected according to a selection criterion considering the change within the problem's objective function caused by a move.
3. At the end of a local search algorithm we return the best solution that has been found by the local search algorithm.

Considering these basic steps of a local search algorithm we identify the following four basic building blocks of local search algorithms: an initial solution, a set of moves, which are applied to a current solution in order to compute a local neighborhood, an objective function, and a selection criterion choosing a move to obtain the next solution.

Design Decision. TEMPLE must provide abstractions and notations to describe how to compute an *initial solution* and *moves* for a particular staff scheduling problem instance. Moreover, we must be able to define a problem's *objective function* in TEMPLE. Selection criteria should not be a part of the TEMPLE modeling language, to keep it as simple as possible.

5.3 The TEMPLE Modeling Language

5.3.1 Interval Declaration

To model a particular resource planning and staff scheduling problem we must declare the different kinds of intervals a problem consists of. Each interval has four basic properties: `Start`, `Duration`, `End`, and a boolean basic property `Active`, indicating whether an interval is part of a problem's solution or not. If necessary, we can further define additional basic properties for intervals. For instance, in the following code sample we declare that a staff scheduling problem consists of shifts, breaks and time slots, the latter having an additional property modeling staffing requirements:

```
Interval Shift;
Interval Break;

Interval TimeSlot with StaffingRequirement;
```

5.3.2 Links Declaration

In TEMPLE links between intervals are declared by using arrows or the keyword `contains` in the following manner:

```
//Declaration of a uni-directional link:
//Each shift is linked to zero or several breaks but NOT vice versa.
Shift  ->  Break;


//Declaration of a bi-directional link:
//Each shift is linked to zero or several breaks and vice versa.
Shift  <->  Break;


//Declaration of a bi-directional link:
//Each shift is linked to zero or several breaks and vice versa.
//If a shift is not active its associated breaks are also inactive.
Shift  contains  Break;


//Declaration of a bi-directional link with rolenames:
Employee[Trainer]  <->  Employee[Trainee];
```

5.3.3 Derived Properties

In TEMPLE, we can derive additional interval properties on the basis of basic properties or previously defined ones. For instance, to derive a property reflecting the total break time scheduled in a shift, as shown in Figure 5.4, we have to insert the following lines of code into a TEMPLE program:

```
Property Shift::TotalBreakTime( Shift.Break[] scheduledBreak)
{
  TotalBreakTime = sum(i in scheduledBreak.getRange()) (scheduledBreak[i].Duration);
}
```

This code snippet specifies that each shift has an additional property called `TotalBreakTime`. This property is derived from all breaks linked to a single shift `Shift.Break[]`, which can be accessed through the alias `scheduledBreak`. The value of property `TotalBreakTime` is computed by summing up the durations of each break scheduled within the shift. Similarly, we can also derive a shift's break time percentage:

```
Property<float> Shift::TotalBreakTimeInPercent(Shift thisShift)
{
  TotalBreakTimeInPercent = (thisShift.TotalBreakTime * 100.0 ) / thisShift.Duration;
}
```

Since percentages are not necessarily integer values, we use the tag `<float>` to ensure that a floating point value is used to represent this property.

5.3.4 Derived Constraints

So far we have already specified the two derived properties from Section 5.2.2 and Figure 5.4. To model that example completely we have to insert a constraint on the minimum break time to be scheduled in a shift.

In TEMPLE we distinguish between two kinds of constraints, hard constraints and soft constraints. Hard constraints specify the criteria which must be satisfied completely by any feasible solution. Except for the keyword `HardConstraint` the violation degree of a hard constraint is derived in the same manner as the value of a derived property. The following hard constraint definition checks whether a shift's break time percentage is not below a required twenty percent threshold:

```
HardConstraint Shift::MinimumBreakTime(Shift thisShift)
{
  if(thisShift.TotalBreakTimeInPercent < 20)  MinimumBreakTime = VIOLATED;
}
```

Soft constraints on the other hand model the objectives that shall be minimized by a good solution. The importance of a soft constraint within an entire staff scheduling problem is expressed in terms of integer weights, as shown within the following example:

```
SoftConstraint<float> Shift::MinimumBreakTime(Shift thisShift) weight(10)
{
  MinimumBreakTime = max( 0, 20 - thisShift.TotalBreakTimeInPercent);
}
```

Curve Operation	*Description*
`void Pulse(int start, int end, bool active)`	If `active` is `true` the curve is incremented in each time slot from `start` to `end` by one unit.
`void Pulse(int start, int end, bool active, int value)`	If `active` is `true` the curve is incremented in each time slot from `start` to `end` by `value` units.
`void Value(int position, int value)`	A curve's entry at index `position` is set to `value`.
`int Value(int position)`	Returns the `value` stored in the curve at `position`.
`void Add(Curve otherCurve)`	The values of `otherCurve` are added to the curve.
`void Subtract(Curve otherCurve)`	The values of `otherCurve` are subtracted from the curve.
`void CyclicAdd(Curve otherCurve, int cycleLength)`	The values of `otherCurve` are added to the curve. A value at position `i` in `otherCurve` is added to the value at position `i % cycleLength` in the curve. This operation is used in problems having a cyclic planning period.
`void CyclicSubtract(Curve otherCurve, int cycleLength)`	The values of `otherCurve` are subracted from the curve. A value at position `i` in `otherCurve` is subtracted from the value at position `i % cycleLength` in the curve. This operation is used in problems having a cyclic planning period.
`void AddPositiveValues(Curve otherCurve)`	Only positive values of `otherCurve` are added to the curve.
`void AddNegativeValues(Curve otherCurve)`	Only negative values of `otherCurve` are added to the curve.
`void SubtractPositiveValues(Curve otherCurve)`	Only positive values of `otherCurve` are subtracted from the curve.
`void SubtractNegativeValues(Curve otherCurve)`	Only negative values of `otherCurve` are subtracted to the curve.

Table 5.1: Methods provided by TEMPLE to derive a curve from already existing elements.

5.3.5 Derived Curves

In TEMPLE, we can derive curves from intervals and previously formulated curves, by using a predefined set of curve operations. These operations increment or decrement a curve over a certain period, they write or read a value at a specific position, or they add and subtract other, already existing curves. These methods are described in detail in Table 5.1. For instance, the curve presenting an employee's actual working time, as depicted in Figure 5.5, can be modeled in the following way:

```
Curve Shift::WorkingTimePattern(Shift thisShift, Shift.Break[] scheduledBreak)
{
  //Increment curve from shift start until shift end.
  WorkingTimePattern.Pulse( thisShift.Start, thisShift.End, thisShift.Active);

  //Decrement curve along each break.
  forall(i in scheduledBreak.getRange())
  {
    WorkingTimePattern.Pulse( scheduledBreak[i].Start,
                              scheduledBreak[i].End,
                              scheduledBreak[i].Active,
                              -1 );
  }
}
```

5.3.6 Initial Solution

After we have modeled the structure of a particular staff scheduling problem by the help of intervals, links, derived properties, curves and constraints, we have to specify an initial solution for a particular staff scheduling problem. In TEMPLE, the initial solution is formulated in three different steps. First of all, in each TEMPLE program we specify an input XML-file. That XML-file contains a list of intervals and stores the initial basic properties of each interval. This initialization step as well as the underlying XML-format will be described in detail at a later point in Section 7.2. Secondly, we can force the instantiation of further intervals. To do so we define how many intervals of one type are instantiated for each interval of another kind. For instance, the following lines of code cause the instantiation of four breaks in each shift:

```
Instantiate Shift.Break[] ()
{
  Shift.Break[].Count = 4;
}
```

Thirdly, we may compute and assign initial values to the basic interval properties `Start`, `Duration`, and `Active`. The initial values are derived from already existing properties or curves of linked intervals. Furthermore, we can also restrict the domains of basic properties and we can introduce additional links between intervals:

```
Initialize Shift::BreakSchedule(Shift thisShift, Shift.Break[] breakToSchedule)
{
  forall(i in breakToSchedule.getRange())
  {
    //1. Assign initial values to basic break properties
    breakToSchedule[i].Start    = thisShift.Start;
    breakToSchedule[i].Duration = 30 minutes;
    breakToSchedule[i].Active   = true;

    //2. Restrict the variable domain of a break's start and its duration
    forall(j in thisShift.Start .. thisShift.End)
      breakToSchedule[i].Start.Domain.Add(j);

    breakToScheduled[i].Duration.Domain.Add(30 minutes);

    //3. Link the shift with each break scheduled within it.
    breakToSchedule[i].AddLink(thisShift, "Shift");
  }
}
```

5.3.7 Moves

To define moves in TEMPLE we must compute and assign new values to basic interval properties. For instance, the following code snippet specifies a move placing a break at a new, randomly chosen, position in its associated shift:

```
Move Shift::PutBreakAtNewPosition(Shift thisShift, Shift.Break[] scheduledBreak)
{
  range S = thisShift.Start .. thisShift.End;

  select(i in scheduledBreak.getRange())
    select(newPosition in S)  scheduledBreak[i].Start = newPosition;
}
```

5.3.8 Further Language Details

For the sake of completeness, we describe which additional information must be specified to obtain a compilable TEMPLE program: an input XML-file containing input intervals and initial basic property values, a solution XML-file in which the obtained solution of a problem shall be saved, the local search algorithm which shall be applied to a particular problem, a limit on the algorithm running time and the granularity of the planning period.

```
input                  = "./input_data.xml";
solution               = "./solution.xml";
algorithm              = iterated local search;
algorithm running time  = 1 minute;
time slot              = 10 minutes;
```

5.3.9 Optimization Goal and Objective Function

In TEMPLE we use hard and soft constraints to define the optimization goals of a considered staff scheduling problem. If S denotes the set of all soft constraints and H the set of all hard constraints defined in a particular TEMPLE program, the local search algorithms generated by our TEMPLE compiler try to solve the following optimization problem:

$$min \quad \sum_{s \in S} s.Weight \times s.ViolationDegree$$

$$s.t. \quad \forall h \in H : h.ViolationDegree = 0$$

5.4 A First TEMPLE Model

In this section we consider a small toy example to derive a first complete TEMPLE model. Despite its conciseness, the considered problem has all common characteristics of staff scheduling problems. To solve it we will use all language elements provided by TEMPLE. As input for the sample staff scheduling problem we are given:

1. The staffing requirements over a planning period from 06:00 until 17:00. The planning period is divided into 54 time slots of ten minutes length. The staffing demand r_t for time slot t requires that during the t-th time slot at least r_t employees must be working.
2. A sample shift plan consisting of three shifts all having the same duration of 8 hours.

Our optimization goal is to schedule breaks in each shift such that the five following requirements hold:

$Requirement_1$: A single break must last 30 minutes at least.

$Requirement_2$: Breaks must not be placed outside a shift.

$Requirement_3$: Each shift must contain at least 25% break time.

$Requirement_4$: Two distinct breaks must not overlap with each other.

$Requirement_5$: In each point of time the staffing requirements must be satisfied completely.

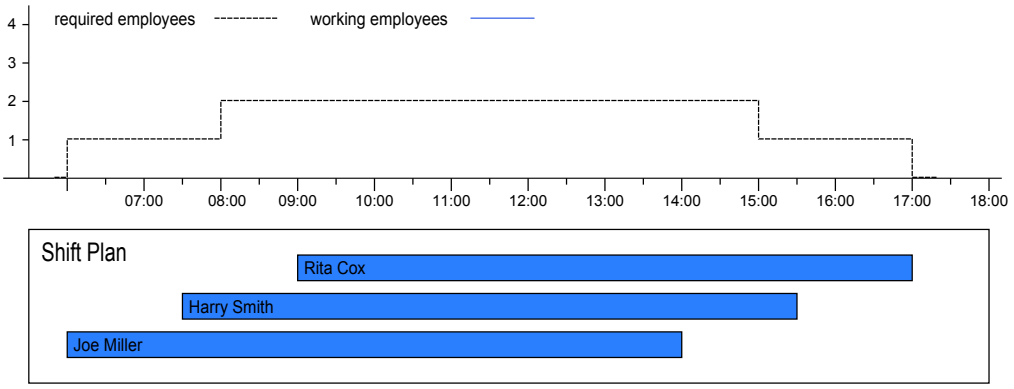


Figure 5.6: Problem input for our sample resource planning and staff scheduling problem.

Figure 5.6 depicts the input shift plan and staffing requirements for our small sample staff scheduling task. The staffing requirements were chosen in such a manner that they can be satisfied completely if the percentage of break time scheduled in each shift is exactly 25%. All experiments in the remainder of this sections will be carried out on a Genuine Intel T2400 laptop running at 1.8 GHz with 2 Gbytes of RAM.

5.4.1 Intervals and Links

At first glance, our sample problem consists of three kinds of intervals: shifts, breaks and time slots. In addition to the basic properties of intervals, `Start`, `Duration`, `End`, and `Active`, each time slot has an additional property called `StaffingRequirement`. This extra property encodes the number of employees that should be working between a time slot's `Start` and its `End`. Further, we introduce a single interval named Problem into model for our sample task. Problem acts as a kind of root interval to which properties, curves and constraints regarding the entire problem will be associated. These considerations lead to the following interval declarations in our first TEMPLE program:

```
Interval Problem;
Interval Shift;
Interval Break;
Interval TimeSlot   with StaffingRequirement;
```

Considering the relations between time intervals, there is obviously a link between a shift and the breaks placed within it and vice versa. Since the entire problem consists

of several shifts and time slots we also connect the single problem root interval with each shift and time slot.

```
Problem      -> Shift;
Shift       <-> Break;
Problem      -> TimeSlot;
```

5.4.2 The First Constraints

Right now, we are already able to define our first constraint on a feasible solution. Each break is required to last at least 30 minutes. Consequently, for each break we introduce a hard constraint called `MinimumDuration`, which depends solely on the break itself. This hard constraint is violated whenever a break is shorter than 30 minutes:

```
//Requirement 1: A single break must last 30 minutes at least.
HardConstraint Break::MinimumDuration(Break thisBreak)
{
  if(thisBreak.Duration < 30 minutes)   MinimumDuration = VIOLATED;
}
```

In the same manner we ensure that breaks must not be scheduled outside a shift. For each shift we impose a hard constraint named `ScheduleBreaksWithinShift`, depending on the shift itself and all breaks scheduled within it. The constraint `ScheduleBreaksWithinShift` is violated if a break starts before or ends after the shift it is scheduled within:

```
//Requirement 2: Breaks must not be placed outside a shift.
HardConstraint Shift::ScheduleBreaksWithinShift(Shift thisShift, Shift.Break[] scheduledBreak)
{
  forall(i in scheduledBreak.getRange())
  {
    if(scheduledBreak[i].Start < thisShift.Start)
      ScheduleBreaksWithinShift = VIOLATED;

    if(thisShift.End < scheduledBreak[i].End)
      ScheduleBreaksWithinShift  = VIOLATED;
  }
}
```

5.4.3 Properties

So far, we have already specified two hard constraints directly. Let's consider the requirement that each shift must contain not less than 25% break time. We are now going to define this constraint in several steps. First of all, for each shift we derive a property `TotalBreakTime` encoding a shift's total amount of break time:

```
Property Shift::TotalBreakTime( Shift.Break[] scheduledBreak)
{
  TotalBreakTime = sum(i in scheduledBreak.getRange()) (scheduledBreak[i].Duration);
}
```

Secondly, for each shift we define another derived property named `TotalBreakTimeInPercent`, which simply converts a shift's absolute amount of break time into the corresponding percentage value. Note, that since a percentage needs not to be an integer value the property is declared to be of type float:

```
Property<float> Shift::TotalBreakTimeInPercent(Shift thisShift)
{
  TotalBreakTimeInPercent = (thisShift.TotalBreakTime * 100.0 ) / thisShift.Duration;
}
```

Finally, for each shift we impose a hard constraint, which requires that a shift's percentage of break time must be at least 25%:

```
//Requirement 3: Each shift must contain at least 25 % break time.
HardConstraint Shift::MinimumBreakTime(Shift thisShift)
{
  if(thisShift.TotalBreakTimeInPercent < 25)  MinimumBreakTime = VIOLATED;
}
```

5.4.4 Curves

The next requirement we consider requires that two distinct breaks must not overlap with each other. One possible way of modeling that constraint might be to check for any pair of distinct breaks if they overlap in time. However, at this point we choose a different approach. In an intermediate step we use a curve to model a shift's break pattern. For each shift we introduce a curve `BreakPattern` that is derived from the breaks scheduled within the shift:

```
Curve Shift::BreakPattern(Shift.Break[] scheduledBreak)
{
  forall(i in scheduledBreak.getRange())
    BreakPattern.Pulse( scheduledBreak[i].Start,
                        scheduledBreak[i].End,
                        scheduledBreak[i].Active);
}
```

If two or more breaks overlap at a certain point of time, the curve value is incremented several times, resulting in a value strictly greater than one. Thus the hard constraint `NoOverlappingBreaks`, requiring that breaks do not overlap with each other, is violated as soon as the break pattern curve contains a value greater than one:

```
//Requirement 4: Two distinct breaks must not overlap with each other.
HardConstraint Shift::NoOverlappingBreaks(Shift thisShift)
{
  Curve breakPattern = thisShift.BreakPattern;

  forall(i in breakPattern.Period())
    if(breakPattern.Value(i) > 1)
      NoOverlappingBreaks = VIOLATED;
}
```

5.4.5 The Complete Problem Model

There is only one requirement left to be defined in TEMPLE. At any time, the number of working employees must not under-run the staffing requirements. Again, we use a multi-step approach to model this constraint. Firstly, for the entire problem we derive a curve named `StaffingRequirements`. This curve reflects the staffing requirements over the entire planning period.

```
Curve Problem::StaffingRequirements(Problem.TimeSlot[] timeSlot)
{
  forall(i in timeSlot.getRange())
    StaffingRequirements.Pulse( timeSlot[i].Start,
                                timeSlot[i].End,
                                timeSlot[i].Active,
                                timeSlot[i].StaffingRequirement);
}
```

Secondly, we derive a curve encoding the number of working employees in each time slot. This curve is obtained by summing up all available employees and subtracting each employee's break time.

```
Curve Problem::WorkingStaff(Problem.Shift[] scheduledShift)
{
  forall(i in scheduledShift.getRange())
  {
    WorkingStaff.Pulse    ( scheduledShift[i].Start,
                            scheduledShift[i].End,
                            scheduledShift[i].Active);

    WorkingStaff.Subtract ( scheduledShift[i].BreakPattern);
  }
}
```

Thirdly, by subtracting the curve representing staffing requirements from the curve reflecting the actually working employees, we obtain a curve `DeviationFromStaffingRequirements` that gives us the deviation of staffing requirements in each time slot.

```
Curve Problem::DeviationFromStaffingRequirements(Problem thisProblem)
{
  DeviationFromStaffingRequirements.Add     (thisProblem.WorkingStaff);
  DeviationFromStaffingRequirements.Subtract(thisProblem.StaffingRequirements);
}
```

A positive curve value at a point of time indicates that more employees than required are working, a negative curve value reports shortage of staff. According to our problem definition, we are only interested in avoiding shortage of employees. Therefore we derive another curve `Shortage` from `DeviationFromStaffingRequirements` which highlights only understaffed time slots in our planning period.

```
Curve Problem::Shortage(Problem thisProblem)
{
  Shortage.SubtractNegativeValues(thisProblem.DeviationFromStaffingRequirements);
}
```

Finally, we impose the soft constraint `NoShortage` to reduce shortage of employees during optimization. The violation degree of soft constraint `NoShortage` is obtained by summing all values from curve `DeviationFromStaffingRequirements`.

```
Requirement 5: In each point of time the staffing requirements must be satisfied completely.
SoftConstraint Problem::NoShortage(Problem thisProblem)
{
  Curve      shortage   = thisProblem.Shortage;
  NoShortage            = sum(i in shortage.Period()) (shortage.Value(i));
}
```

5.4.6 Initial Solution

Right now we have succeeded to model each requirement of our sample resource planning and scheduling problem, by introducing four hard constraints and one soft constraint. In the next two major steps we have to provide:

- ▷ The number of breaks to be scheduled in each shift.
- ▷ An initial feasible break schedule that satisfies all hard constraints of our problem model.

To compute the number of breaks to be scheduled in each shift we again rely on a step-wise approach. For each shift we derive a property named `RequiredBreakTime` which is the minimum amount of break time above the required 25% threshold.

```
Property Shift::RequiredBreakTime(Shift thisShift)
{
  RequiredBreakTime = (int) ceil((thisShift.Duration * BREAK_TIME_PERCENTAGE) / 100.0);
}
```

Then, for each shift we derive the number of (30-minute) breaks that must be scheduled in order to exceed the minimum amount of break time.

```
Property Shift::NumberOfBreaks(Shift thisShift)
{
  float breakTimeToSchedule  = thisShift.RequiredBreakTime;

  NumberOfBreaks  += (int) ceil(breakTimeToSchedule / MINIMUM_BREAK_DURATION);
}
```

Finally, we specify that the recently computed number of (30-minute) breaks is scheduled per each shift.

```
Instantiate Shift.Break[] (Shift thisShift)
{
  Shift.Break[].Count = thisShift.NumberOfBreaks;
}
```

In TEMPLE, all hard constraints must be satisfied by the initial solution obtained for a particular problem instance. The information on how to compute a feasible initial solution must be provided by the user. With regard to our sample problem, we must place the breaks in each shift such that we create a legal break pattern, i.e., the break pattern is consistent with all hard constraints. For that purpose we set each break's duration to be 30 minutes. In that manner we guarantee the minimum break time percentage as well as the required minimum break duration. The first break is scheduled one hour after shift start. All other breaks start one hour after their predecessor break has started. Since each break lasts 30 minutes, we can ensure that there is at least half an hour between each break and they do not overlap. Moreover, breaks are scheduled entirely in their corresponding shifts. We further restrict the domain for each break start to lie within the start and end of the corresponding shift, and we link each break to the shift it is scheduled within.

```
Initialize Shift::BreakSchedule(Shift thisShift, Shift.Break[] breakToSchedule)
{
  range S = thisShift.Start         .. thisShift.End;

  forall(i in breakToSchedule.getRange())
  {
    breakToSchedule[i].Start     = thisShift.Start +  i * 1 hour;
    breakToSchedule[i].Duration  = 30 minutes;
    breakToSchedule[i].Active    = 1;

    breakToSchedule[i].Start.Domain.Clear();

    forall(j in S)  breakToSchedule[i].Start.Domain.Add(j);

    breakToSchedule[i].AddLink(thisShift, "Shift");
  }

}
```

5.4.7 Moves

For our sample problem it suffices to specify only one single move, that will be applied iteratively to improve the quality of an incumbent solution. This single move is defined for each shift and it is named `PutBreakAtNewPosition`. The move selects a break and a position within the shift at random and moves the break to that newly selected position.

```
Move Shift::PutBreakAtNewPosition(Shift thisShift, Shift.Break[] scheduledBreak)
{
  range S = thisShift.Start .. thisShift.End;

  select(i in scheduledBreak.getRange())
    select(newPosition in S)  scheduledBreak[i].Start = newPosition;
}
```

5.4.8 Solving the Problem

Now we have nearly finished our first TEMPLE program. Before running it we add information on the input XML-file containing the staffing requirements and shift plan and we specify the file in which the best solution found will be stored. As local search strategy we select iterated local search and impose a one-minute running time limit to it. Finally, we declare that our planning period is divided into 10-minute time slots.

```
input                  = ".\Example-1-input.xml";
solution               = ".\Example-1-output.xml";
algorithm              = iterated local search;
algorithm running time = 1 minute;
time slot              = 10 minutes;
```

Finally, we invoke the TEMPLE compiler, which will be described in detail in Chapter 7, to translate our TEMPLE program into executable code. The obtained local search algorithms terminates within one second and returns a solution which satisfies all hard constraints and avoids shortage of staff with respect to staffing requirements. Figure 5.7 shows the obtained break schedule. Each break last exactly 30 minutes, is scheduled within its associated shift, and does not overlap with any other break. Each of the eight-hour shifts contains four 30-minute breaks, having two hours of break time in total which is exactly the required 25% required. Moreover, at each point in time there are exactly as many employees working as demanded by the staffing requirements. Thus, we conclude that TEMPLE was able to compute a feasible solution with optimal quality.

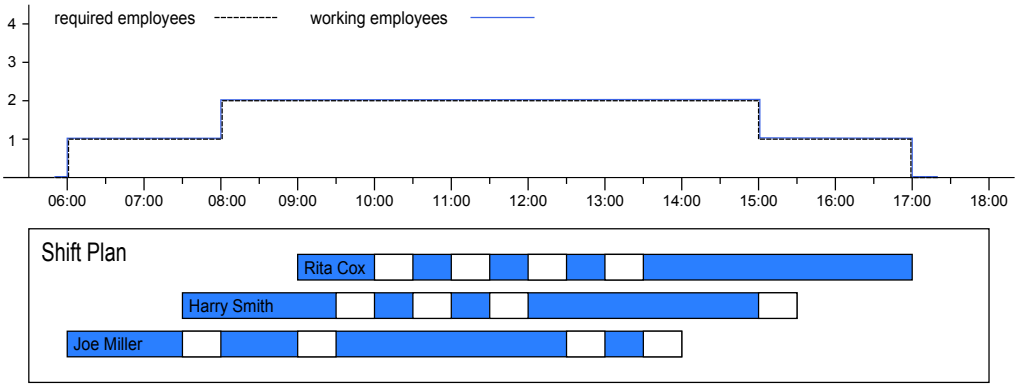


Figure 5.7: Solution obtained with our TEMPLE program for our sample resource planning and scheduling problem.

5.4.9 An Extended Problem

In the solution for our sample problem shown in Figure 5.7 breaks are scheduled very irregularly in each shift. It might occur, that employees work for three hours or even longer without having a break. This can lead to stress and exhaustion which must be avoided by a reasonably designed break schedule. Consequently, we will extend our TEMPLE program to obtain a break schedule in which employees do not work longer than 100 minutes without having a break. Thereby, we demonstrate that, thanks to their modular style, TEMPLE programs can be modified and extended easily, to react quickly to changes in requirements or user needs.

First of all, for each break we derive a property computing the distance between the

break and its predecessor. If a break is the first break of a shift, the property will have a value of zero:

```
Property Break::DistanceToPredecessor(Break thisBreak, Break.Shift().Break() allBreaksInShift)
{
  //Determine the index of predecessor break
  selectMax(i in allBreaksInShift.getRange() :
                allBreaksInShift[i].End <= thisBreak.Start)   (allBreaksInShift[i].End)
  {
    //Compute distance to predecessor.
    DistanceToPredecessor = thisBreak.Start - allBreaksInShift[i].End;
  }
}
```

Secondly, for each shift we introduce a property measuring the time elapsing from a shift's start until the first break of the shift:

```
Property Shift::DistanceToFirstBreak(Shift thisShift, Shift.Break[] scheduledBreak)
{
  //Determine first break in shift
  selectMin(i in scheduledBreak.getRange()) (scheduledBreak[i].Start)
  {
    DistanceToFirstBreak = scheduledBreak[i].Start - thisShift.Start;
  }
}
```

Thirdly, we introduce a property modeling the time between a shift's last break and the shift end:

```
Property Shift::DistanceToLastBreak(Shift thisShift, Shift.Break[] scheduledBreak)
{
  //Determine last break in shift
  selectMax(i in scheduledBreak.getRange()) (scheduledBreak[i].End)
  {
    DistanceToLastBreak = thisShift.End - scheduledBreak[i].End;
  }
}
```

Finally, we are able to derive a soft constraint which requires that employees should not work longer than 100 minutes in a row. This soft constraint regards the duration of each working period in the shift. If a working period lasts longer then 100 minutes, the deviation from 100 minutes is added to the soft constraint's violation degree:

```
//Additional requirement: Employees should not work longer than 100 minutes in a row.
SoftConstraint Shift::WorkingPeriodDuration(Shift thisShift, Shift.Break[] scheduledBreak)
{
  WorkingPeriodDuration  += max(thisShift.DistanceToFirstBreak - 100 minutes, 0);
  WorkingPeriodDuration  += max(thisShift.DistanceToLastBreak  - 100 minutes, 0);

  forall(i in scheduledBreak.getRange())
    WorkingPeriodDuration  += max(scheduledBreak[i].DistanceToPredecessor - 100 minutes, 0);
}
```

The local search algorithm obtained from the extended TEMPLE program returns an optimal solution after eight seconds. Figure 5.8 shows the obtained break schedule. We observe that our additional restriction on working periods is completely satisfied.

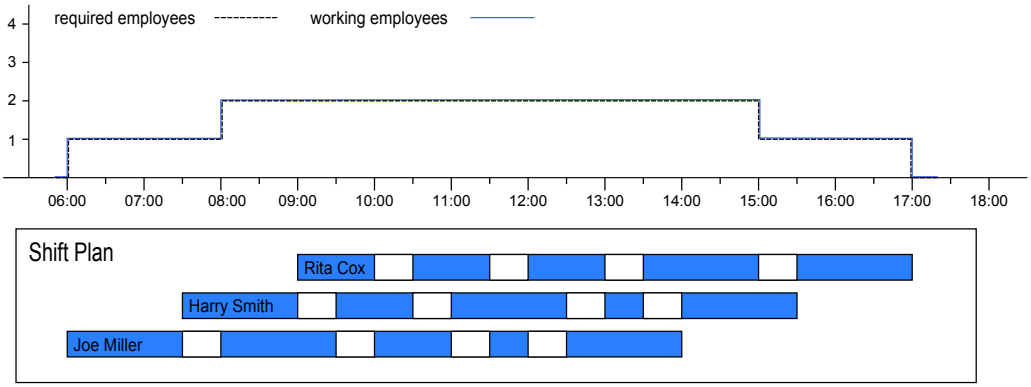


Figure 5.8: In this solution for our sample problem no working period lasts longer than 100 minutes.

In a last revision step we will extend our TEMPLE problem even further. It is common in many companies to assign lunch breaks to their employees, and these lunch breaks usually last longer than ordinary breaks. We extend our sample problem by an additional hard constraint, which requires that each shift must contain a one-hour lunch break. Not only does this additional criterion require to insert additional properties and constraints, but it has also side-effects to already existing elements within our TEMPLE program. Once again, we will see, that we will extend and modify our existing TEMPLE program by adding and changing only a few lines of code, thus, TEMPLE programs are very robust against changes, as they frequently occur in any daily working area.

First of all, for each break we insert an additional property specifying whether a break is a lunch break or not. Thus property will be set to true if a break reaches or exceeds the limit of sixty minutes required for a lunch break.

```
Property Break::IsLunchBreak(Break thisBreak)
{
  if(thisBreak.Duration >= 60 minutes)    IsLunchBreak = true;
}
```

Secondly, for each shift we derive a property indicating the number of lunch breaks scheduled per shift.

```
Property Shift::LunchBreakCount(Shift.Break[] scheduledBreak)
{
  forall(i in scheduledBreak.getRange() : scheduledBreak[i].IsLunchBreak == true)
    LunchBreakCount++;
}
```

Finally, for each shift we introduce a hard constraint, which is violated whenever a shift lacks a lunch break.

```
//Additional requirement: A shift must contain at least one lunch break.
HardConstraint Shift::LunchBreak(Shift thisShift)
{
  if(thisShift.LunchBreakCount < 1)    LunchBreak = VIOLATED;
}
```

By introducing additional properties and a hard constraint we are not done yet. As we remember when scheduling an initial break pattern in a shift we have only used 30-minutes breaks. When scheduling an initial break pattern we must guarantee that at least one break's duration is set to 60 minutes. Fortunately, this change can be implemented by inserting only one additional line of code into our original initialization block. Among all breaks we select a break at random and extend its duration to 60 minutes, thus, making it a lunch break.

```
Initialize Shift::BreakSchedule(Shift thisShift, Shift.Break[] breakToSchedule)
{
  range S = thisShift.Start         .. thisShift.End;

  forall(i in breakToSchedule.getRange())
  {
    breakToSchedule[i].Start     = thisShift.Start +  i * 1 hour;
    breakToSchedule[i].Duration  = 30 minutes;
    breakToSchedule[i].Active    = 1;

    breakToSchedule[i].Start.Domain.Clear();

    forall(j in S)  breakToSchedule[i].Start.Domain.Add(j);

    breakToSchedule[i].AddLink(thisShift, "Shift");
  }

  //Select an arbitrary break and make it a lunch break
  select(i in breakToSchedule.getRange()) breakToSchedule[i].Duration = 60 minutes;
}
```

Furthermore, we must also modify the code for property `Shift::NumberOfBreaks`. In two intermediate steps we set the number of breaks that will be created per shift to one and subtract 60 minutes, the duration for the lunch break, from the break time to be scheduled. The remaining break time is then distributed among 30-minute breaks.

```
Property Shift::NumberOfBreaks(Shift thisShift)
{
  float breakTimeToSchedule  = thisShift.RequiredBreakTime;

  NumberOfBreaks                =  1; //Create a lunch break
  breakTimeToSchedule          -= 60 minutes; //Subtract lunch break time

  //Distribute remaining break time among 30-minute breaks
  NumberOfBreaks               += (int) ceil(breakTimeToSchedule / 30 minutes);
}
```

So far, we added an additional hard constraint requiring a one-hour lunch break per shift to our problem model, and we modified the TEMPLE code effecting how an initial feasible break schedule is computed for our problem. Since from now, shifts contain breaks of different duration, 30 minutes, and 60 minutes, we introduce an additional move for our local search algorithm which swaps lunch break with an ordinary 30-minute break.

```
Move Shift::SwapTwoBreaks(Shift thisShift, Shift.Break[] scheduledBreak)
{
  select(i in scheduledBreak.getRange())
  {
    select(j in scheduledBreak.getRange() :
                scheduledBreak[i].Duration != scheduledBreak[j].Duration)
    {
      int t                     = scheduledBreak[i].Start;
      scheduledBreak[i].Start   = scheduledBreak[j].Start;
      scheduledBreak[j].Start   = t;
    }
  }
}
```

After recompiling our TEMPLE program the iterated local search algorithm is able to return an optimal solution to our small problem after 40 seconds. Figure 5.9 shows the obtained break schedule containing a 60-minute lunch break in each shift.

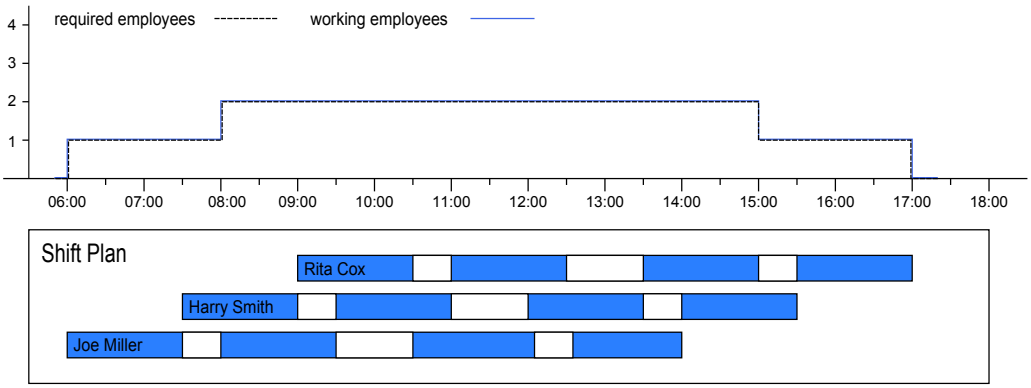


Figure 5.9: Solution for our sample problem, in which each shift contains one 60-minute lunch break.

Chapter 6

Related Work

In this chapter we review state-of-the-art modeling languages and metaheuristic frameworks aimed at scheduling tasks or general combinatorial optimization problems. In particular, we examine whether the basic building blocks of staff scheduling problems and local search algorithms, are supported within these approaches. Moreover, we analyze if our design goals for a modeling language for staffs scheduling problems from Section 5.1 can be realized by these approaches.

6.1 Related Modeling Languages

6.1.1 ESRA - An Executable Symbolism for Relational Algebra

Flener et al. developed the language ESRA [56], an Executable Symbolism for Relational Algebra, to model combinatorial optimization problems on the basis of sets, entities and relations. This approach has been successfully applied in general modeling and specification languages like ALLOY [34], the Object Constraint Language (OCL) [57] of the Unified Modeling Language (UML) [49], or in entity relationship (ER) diagrams.

In ESRA combinatorial optimization problems are specified by declaring *domains*, *constants* and *decision variables* involved in the considered task. For optimization problems we also have to define a *cost function* which shall be minimized or maximized. Domains, constants and decision variables are declared on the basis of sets and enumeration types, and complex data types. Complex data types are defined by using relations and cardinalities between simpler data types. The constraints imposed on a particular problem are formulated in first order logic. ESRA provides a fixed set of predicates and functions which can be used within constraint formulations. Figure 6.1 presents an ESRA model of the traveling salesman problem taken from [56].

`dom` *Cities*	
`cst` *Distance*:	$(Cities \times Cities) \rightarrow \mathbb{N}$
`var` *Next*:	$Cities \rightarrow^1 Cities$
`minimise`:	$\sum_{c \in Cities} Distance(c, Next(c))$
`such that`:	$\forall\ (c_1 \wedge c_2 \in Cities)\ Next^*(c_1) = c_2$

Figure 6.1: ESRA model of the traveling salesman problem [56].

6.1.2 ESSENCE

ESSENCE [22] is a *problem specification language* for combinatorial problems. The main motivation for ESSENCE was to create a language in which both decision problems and optimization problems can be specified in a very natural, concise way at a very abstract level. The formal problem specification shall then be mapped automatically to a constraint satisfaction problem (CSP) model and finally be solved by a corresponding solver. According to Frisch et al. [22], three main goals were realized in ESSENCE:

1. ESSENCE is a very natural language that is understandable to anyone having basic knowledge in discrete mathematics. No background in constraint programming is required by a potential user.
2. ESSENCE provides a high level of abstraction. A few statements are sufficient to specify a combinatorial problem.
3. A problem specified in ESSENCE can be effectively mapped to constraint satisfaction problems (CSPs).

ESSENCE specifications a very similar to the those specifications of combinatorial problems given by Johnson and Garey [24]. An ESSENCE specification consists of seven kinds of statements, each of which starting with one of the following keywords: `given`, `where`, `letting`, `find`, `minimising`, `maximising` and `such that`. `given` statements are used to specify the input parameters of a combinatorial problem. `where` statements define allowed input parameter values. `letting` statements introduce constant identifiers and user defined types. `find` statements are used to declare the decision variables of a combinatorial problem. `minimising` and `maximising` statements are used to define the objective functions of a combinatorial optimization problem. `such that` statements are used to specify the constraints involved in a combinatorial problem.

Figure 6.2 presents an ESSENCE specification of the well-known knapsack problem. In the knapsack problem we are given a set of items, each having a specific weight

and value. To solve the knapsack problem we must find a collection of items such that their total value is as large as possible and their total weight must not exceed a certain upper bound. Considering the ESSENCE specification presented in Figure 6.2 we see that the ESSENCE specification is very similar to the specification of the knapsack problem given in natural language.

`given`	U `enum` (...), $w : U \rightarrow$ `int` (...), $v : U \rightarrow$ `int` (...), B : `int`	Given a set of items each of which having a specific weight and value and given limit on the total weight
`find`	U' : `set of` U	find a collection of items
`maximising`	$\sum_{u \in U'} v(u)$	of maximum value
`such that`	$\sum_{u \in U'} w(u) \leq B$	such that the limit on the total weight is not exceeded.

Figure 6.2: ESSENCE specification of the knapsack problem formulated as optimization problem [22].

Frisch et al. [22] report that a suite of 58 problems, 26 drawn from CSPLib, 32 from the literature, could be specified successfully in ESSENCE by an undergraduate student in computer science having no background on constraint programming. Moreover Frisch et al. implemented a rule-base system called CONJURE that can translate a fragment of the ESSENCE language into a constraint programming model. These models were further mapped to ECL^{i}PS^e [55] and Minion [28]. For future work Frisch et al. want to translate the complete ESSENCE language into constraint satisfaction problem models and in that manner they plan to make a significant steps forward into the directions of fully-automated modeling.

6.1.3 The Zinc Modeling Language

Zinc [38] is a high-level modeling language for combinatorial optimization problems. Zinc is a declarative, functional language using a mathematical-like notation. The Zinc language has been designed as simple as possible, offering only a manageable amount of data types, predefined predicates, functions and constraints. However, in contrast to many other specification or modeling languages, e.g., ESRA and ESSENCE, Zinc enables the definition of user-defined predicates, functions, and constraints. With these user-defined language constructs the Zinc language can be extended and adapted to new application domains.

The most important design goal of Zinc was that Zinc must be a solver independent modeling language. Each problem formulated in Zinc can be transformed into a constraint programming model, a mixed integer programming model, or a local search based model. Afterwards developers can experiment with different, already existing, optimizations algorithms and they may chose the solving technique which proved to perform best for a given problem instance.

Rafeh et al. successfully implemented a Zinc compiler [47] which translates a given Zinc program into an intermediate representation in the slightly different language flattened Zinc. This intermediate model is finally transformed into an executable constraint programming model, mixed integer programming model, and local search model by applying appropriate rewriting rules.

With their Zinc compiler Rafeh et al. delivered a prove of concept that solver independent modeling is indeed feasible in Zinc. They formulated a series of well-known combinatorial optimization problems in Zinc: the minimisation of open stacks problem (MOSP), the social golfers problem, perfect squares, the N-queens problem, the knapsack problem, job shop scheduling and the production scheduling problem. Each Zinc formulation of these problems could be transformed into a design model of different solving techniques. A comparison with equivalent, manually written problem models, revealed that the models automatically generated by the Zinc compiler needed only a little bit more running time to achieve the same results. Therefore, Rafeh et al. conclude that Zinc introduces only a negligible computational overhead.

6.1.4 OPL - The Optimization Programming Language

As indicated by its name, the Optimization Programming Language (OPL) [31] was developed to model and solve optimization problems. OPL attempts to combine the advantages of mathematical modeling languages and constraint programming languages:

- ▷ Mathematical modeling languages, such as AMPL [21] and GAMS [7], provide high-level algebraic and set notations and they enable the formulation of concise problem models. Moreover, mathematical modeling can be applied by a wide audience, because users formulate constraints solely by the help of equations and inequation and no information concerning the optimization process must be specified.
- ▷ On the other hand constraint programming languages, such us CHIP [17] and OZ [33], provide logical, high-order and global constraints and they allow a user to affect the way the solution space is explored by specifying search procedures.

An industrial implementation of OPL has been realized within the software IBM ILOG CPLEX Optimization Studio. However, in IBM ILOG CPLEX Optimization Studio only those parts of OPL concerning the modeling of problems have been implemented, the abstractions and notations concerning search procedures are not available in that industrial implementation. Problem models obtained via OPL are solved either by the IBM ILOG CPLEX Optimizer engine (for mathematical programming models) or by the IBM ILOG CPLEX CP Optimizer engine (for constraint programming models).

As an additional feature IBM ILOG OPL provides further language elements to facilitate the development of scheduling models. In this respect IBM ILOG OPL is closely related to TEMPLE, thus, we will present the basic scheduling building blocks of IBM ILOG OPL in more detail:

Time intervals. In addition to ordinary decision variables an IBM ILOG OPL scheduling model contains time intervals. Like decision variables also time intervals are subject to optimization, i.e., start or duration of time intervals are variable and consequently positions and durations of time intervals are changed during the search process. In a scheduling model, time intervals usually represent activities or tasks.

Cumulative functions. In IBM ILOG OPL, a cumulative function is a function representing the sum of individual contributions of intervals. Usually, cumulative functions are used to model the usage or consumption of a specific resource over time, e.g., an interval may increase the value of a cumulative function at its start or over its duration. Cumulative functions are very similar to the concept of derived curves within the domain-specific language TEMPLE.

State functions. In IBM ILOG OPL, state functions are used to represent the evolution of a given feature of the environment over time. The main difference between state functions and cumulative functions is that interval variables have an incremental effect on cumulative functions (increasing or decreasing the function value) whereas they have an absolute effect on state functions (requiring the function value to be equal to a particular state or in a set of possible states).

Temporal constraints. IBM ILOG OPL offers a set of temporal constraints concerning the relationships between two or several time intervals. These constraints comprise precedence constraints regulating the relative positions of intervals, no overlap constraints requiring that intervals are disjoint in time, span constraints ensuring that one interval is contained within the other, and synchronize constraints demanding that two or several intervals start and end at the same time.

Specialized constraints. Specialized constraints are imposed on state functions and cumulative functions. They specify legal upper and lower bounds on the value a cumulative constraint may have during a certain period or they require that the environment must be in a particular state such that a specific task can be performed.

If we compare TEMPLE with IBM ILOG OPL and its scheduling features we recognize some similarities between the two modeling languages. The central element of both languages are time intervals, and curves in TEMPLE are similar to the cumulative functions available in IBM ILOG OPL.

6.1.5 Comet

The programming language Comet [32] was developed to combine the advantages of constraint programming and local search algorithms. Constraint programming represents an elegant way to model complex optimization tasks, involving various different criteria and objectives. In a constraint programming language such a problem can be modeled by imposing each criterion and objective one by another by using logical, high-order and global constraints. However, before Comet was proposed, state of the art constraint solvers had used solely global search algorithms, which more or less explore the entire search space to solve a given optimization task.

Local search algorithms proved to return very good results for many different problems from the literature and from real-life. However, the design of local search algorithms is an art in itself. For instance, a developer must choose an appropriate solution representation, an adequate objective function, and sophisticated data structures to ensure an efficient performance of a local search algorithm. To simplify the design and implementation of local search algorithms, Michel and Van Hentenryck proposed the

modeling language LOCALIZER [40], where local search algorithms can be specified in a manner similar to their pseudo-code representations given in scientific papers.

Later Michel and Van Hentenryck developed a constraint-based architecture for local search algorithms [41]. That architecture included several high-level concepts to model the constraints and objectives of a given optimization problem, and to formulate model-independent local search algorithms applicable to any arbitrary problem model obtained within that architecture. Further Michel and Van Hentenryck presented the constraint-based modeling and optimization language Comet [41] incorporating the proposed architecture. A comprehensive introduction on the Comet language and a detailed overview on the concepts involved in Comet is given in [32].

In the recent past the Comet language has evolved strongly. Besides constraint-based local search algorithms, also constraint programming models and mixed integer linear programs can be specified and solved in Comet. The Comet language and a corresponding just-in-time compiler represent the core of a commercial optimization platform, called the Comet Hybrid Optimization Platform, which is distributed by the company Dynadec Inc (www.dynadec.com).

At this point we want to present the high-level modeling concepts [41] realized in Comet which distinguish it from other modeling languages. These concepts are also used within the code generated by our TEMPLE compiler to obtain an executable program for a specific staff scheduling problem:

Incremental Variables In Comet incremental variables are used to model the dynamic features of the solution to an optimization problem. By dynamic features we understand all aspects of a solution which can be changed during a local search algorithm, e.g., basic decision variables, objective function values or constraint violation degrees.

Invariants Invariants define functional dependencies between incremental variables, and Comet ensures that these functional dependencies are always kept valid during a local search algorithm. For instance, the Comet statement taken from [41],

```
var{int} totalSum(ls) <- sum(i in 1..10) (variableToSum[i]);
```

declares an incremental integer variable `totalSum` and an invariant requiring that `totalSum` is always the summation of `variableToSum[1],...,variableToSum[10]`. `totalSum` is the target variable of that invariant whereas `variableToSum[1],...,variableToSum[10]` are source variables. If a new value is assigned to any source variable `variableToSum[i]` the value of `totalSum` will be updated accordingly so that the functional dependency specified by that invariant is maintained.

Complex Invariants To specify a functional dependency which cannot be declared within a single statement a user must write an entire class file implementing the `Invariant<LS>` interface. Figure 6.3 presents such a user defined invariant ensuring that the value of an incremental variable `totalSum` is always the summation of ten other incremental variables `variableToSum[1],...,variableToSum[10]`. In method `post` we specify that `variableToSum[1],...,variableToSum[10]` are the source variables of that invariant whereas `totalSum` is the single target variable. In method `initPropagation` we compute the sum of all ten source variables and store it in `totalSum`. This method will be executed only once at the start of the local search algorithm written in Comet. Finally, method `propagateInt` updates the target variable, whenever one of the source variables is changed. We compute only the difference in the value of the changed variables and update the sum accordingly. If only a single or a few source variables are changed within an iteration of the local search algorithm, `propagateInt` can be carried out more efficiently than a complete re-computation of the sum from the scratch.

Differentiable Functions In Comet differentiable functions are used to model features of the solution of a given optimization problem. Further they assess the effects that local changes have on these features. Figure 6.4 presents the most relevant methods of interface `Function<LS>` which must be implemented by each specific differentiable function. Each differentiable function maintains the value of a (possibly) complex function, which is computed and maintained by an invariant. The current function value can be accessed at any time through method `value`. Moreover, through method `getAssignDelta` we evaluate the variation of the function value under local changes, i.e., if one or several basic decision variables are assigned new values.

Differentiable Constraints As shown in Figure 6.5, differentiable constraints are very similar to differentiable functions. A differentiable constraint maintains a constraint violation degree which is computed and maintained by an invariant. A differentiable constraint reports whether it is satisfied or violated (method `isTrue`) and it can be queried for its current violation degree (method `violations`). As differentiable functions a differentiable constraint can evaluate the effect of local changes on its violation degree through a method called `getAssignDelta`.

```
class SumOfVariables implements Invariant<LS>
{

  Solver<LS> _ls;
  var{int}[] _variableToSum;
  var{int}   _totalSum;
  bool       _posted;

  //gets incremental variables that shall be summed up
  //and the incremetal variable that shall contain the computed sum.
  SumOfVariables(Solver<LS> ls, var{int}[] variableToSum, var{int} totalSum)
  {

    //passed arguments are stored in local class member variables.

    _ls             = ls;
    _variableToSum  = variableToSum;
    _totalSum       = totalSum;
    _posted         = false;


    //sets source and target variables.
    post();
  }

  //sets source and target variables.
  void post(InvariantPlanner<LS> invariantPlanner)
  {
    if(!_posted)
    {
      //source variables.
      forall(i in 1..10)
        invariantPlanner.addSource(_variableToSum[i]);

      //target variable.
      invariantPlanner.addTarget(_totalSum);
    }
  }

  //computes the sum of source variables and stores it in target variable.
  void initPropagation()
  {
    _totalSum := 0;

    forall(i in 1..10)
      _totalSum := _totalSum + _variableToSum[i];
  }

  //updates target variables efficiently.
  void propagateInt(bool notLastInvocation, var{int} changedVariable)
  {
    //compute change in source variable value.
    int delta  = changedVariable - changedVariable.getOld();

    //update sum by computed change.
    _totalSum := _totalSum + delta;
  }

  ...

}
```

Figure 6.3: Invariant maintaining the sum of several source variables within one target variable.

```
interface Function<LS>
{
  //returns   the current function value.
  var{int}    value();

  //computes the change in function value if variablesToBeChanged are assigned newValuesToBeAssigned.
  int         getAssignDelta  (var{int}[] variablesToBeChanged, int[] newValuesToBeAssigned);

  ...
}
```

Figure 6.4: Interface for differentiable functions.

```
interface Constraint<LS>
{
  //indicates whether a constraint is satisfied or violated.
  var{bool}   isTrue();

  //returns the current constraint violation degree.
  var{int}    violations();

  //computes the change in violation degree if variablesToBeChanged are assigned newValuesToBeAssigned.
  int         getAssignDelta  (var{int}[] variablesToBeChanged, int[] newValuesToBeAssigned);

  ...
}
```

Figure 6.5: Interface for differentiable constraints.

In Comet a problem model of a given optimization is obtained in the following manner:

1. We model the basic decision variables of an optimization problem by the help of incremental variables.
2. We model further features of a problem by using invariants and differentiable functions.
3. Finally, we impose the constraints and objectives of a problem by using invariants and differentiable constraints. All constraints are collected in a constraint system, implementing the differentiable constraint interface.

In Comet local search algorithms operate only with methods provided by differentiable constraints. Consequently, the resulting local search algorithms can be applied to arbitrary problem models consisting of differentiable constraints. Thus, Comet supports the separation of local search algorithms from specific problem models and contributes to the development of general local search algorithms.

6.1.6 ASPEN - An Automated Scheduling and Planning Environment

ASPEN [23] is a modular, reconfigurable application framework which was developed by the Artificial Intelligence Group of the Jet Propulsion Laboratory to model and solve a wide variety of planning and scheduling applications arising at NASA. In particular ASPEN has been applied within the domain of spacecraft operations.

While operating a spacecraft several different high-level science and space craft engineering goals must be achieved. These high-level goals result in a series of low-level spacecraft operations which are required to be scheduled in accordance with a variety of constraints, concerning the availability of resources, the current state of the aircraft and temporal restrictions.

For instance, a high-level goal of a space craft might be to observe and photograph planets, stars, and galaxies. To make a single observation, a series of low-level operations have to be carried out: the space craft must be adjusted, a picture must be taken, the obtained data must be temporarily stored at the space craft and eventually the data must be transmitted to earth. To carry out each single operation successfully one or several conditions need to be satisfied: to take a picture a certain amount of power is required, to save the obtained data enough free storage must be available, and information can be transmitted only at certain down-link times.

The ASPEN Modeling Language

ASPEN provides a modeling language [51] which is used by domain experts to model high-level goals, low-level operations and the constraints imposed on a desired solution. The ASPEN modeling language offers abstractions and notations reflecting these operations and constraints in a very natural way. Consequently, domain experts can formulate planning models very easily and quickly. Afterwards ASPEN schedules the single high-level goals and low-level operations such that as many goals as possible are achieved by the resulting schedule. The scheduling of high-level goals and low-level operations is done automatically without requiring any input from the domain experts. This is particularly useful since the domain experts using ASPEN usually have no knowledge of automated planning and scheduling techniques. The main elements of the ASPEN modeling language are *activities, resources,* and *states*:

Activities. Activities are the central plan elements of ASPEN. Basically, an activity represents a high-level goal or a low-level operation of a specific planning and scheduling problem. A single activity can be decomposed into several subactivities. In that manner a domain expert can model the relation between high-level goals and the low-level operations required to achieve that goal. Each activity has three basic parameters, `start`, `end`, and `duration`, thus an activity can also be considered as a time interval. The value of parameter `duration` usually remains fixed, only the position of an activity is changed during optimization. In additional to `start`, `end`, and `duration`, one can define further parameters for activities.

Resources. ASPEN distinguishes between depletable and non-depletable resources. A depletable resources, e.g. propellent (fuel), is consumed by an activity which uses the resource. A non-depletable resource, e.g. power, is removed from availability only for the duration of an activity. If the activity ends a non-depletable resource will become available again.

States. In ASPEN state timelines are used to represent the evolution of some aspect of a spacecraft over time. A state timeline is associated a set of discrete state values that it can take on, and a list of legal state transitions.

In ASPEN, all constraints in a plan model result from activities. There are four kinds of constraints which activities can impose on other plan elements: *temporal constraints*, *functional dependencies*, *resource reservations*, and *state reservations*:

Temporal constraints. A temporal constraint is a temporal relation between a source activity and a target activity. The relation must be satisfied by every pair of affected activity instances in the plan. The ASPEN language defines six temporal relations:

`starts_before`, `starts_after`, `ends_before`, `ends_after`, `contains` and `contained_by`. A temporal relation can be modified by an optional interval specifying minimum and maximum distances between the pair of activities.

Functional dependencies. Functional dependencies require that the value of a parameter of an activity is a function of other parameter values. In ASPEN, a concrete dependency function itself is written in the programming language `C`.

Resource reservations. In ASPEN the resource requirements of activities are stated by the help of resource reservations. A resource reservation specifies the resource and the number of units needed by an activity along its duration.

State reservations. Activities can impose two kinds of state reservations. A `must_be` reservation requires that the state has a specific value for the duration of the activity. `change_to` reservations change a state to a certain value at the beginning of the activity.

6.1.7 Optimization Algorithms

Usually, the planning and scheduling problems modeled in ASPEN are over-constrained. Therefore, ASPEN tries to obtain a solution maximizing the number of high-level goals which can be achieved without violating any constraints. For that purpose the ASPEN application framework provides three different algorithms for optimizing a specific planning and scheduling task:

1. A greedy, constructive algorithm called forward dispatch.
2. A constructive backtracking algorithm called IRS.
3. A repair-based algorithm.

In practical applications of the ASPEN framework the repair-based algorithm proved to perform best. The repair-based algorithm considers the current solution of a planning and scheduling problem and selects a conflict resulting from the violation of a temporal constraint, a functional dependency, or from a violated resource and state reservation. Then the algorithm tries to resolve that conflict by rescheduling, instantiating and deleting activities, or by assigning new values to parameters in accordance with functional dependencies. A more detailed description on how the repair-based algorithm is realized in ASPEN is given by Rabideau et al. in [46].

	ESRA	ESSENCE	Zinc	ASPEN	OPL	Comet	TEMPLE
Intervals	×	×	×	✓	✓	×	✓
Links	×	×	×	×	×	×	✓
Curves	×	×	×	×	✓	×	✓
Derived Elements	×	×	×	✓	×	✓	✓
Openness	×	×	✓	✓	×	✓	✓
Modularity	×	×	✓	✓	×	✓	✓

Table 6.1: Comparison of TEMPLE and related modeling languages.

6.1.8 Comparison with Temple

To compare the previously presented modeling languages with TEMPLE we examined whether they provide the basic building blocks of staff scheduling problems we identified in Section 5.2: intervals, links, curves, derived properties and constraints. Moreover, we considered if the related modeling languages are open, in the sense of the design goal Openness stated in Section 5.1. In an open modeling language arbitrary aspects of a problem can be modeled and the language is not restricted to a finite set of features and constraints. As a last criterion for the modeling languages related with TEMPLE we examined whether problems can be modeled in a modular manner, as required by out design goal Modulariy in Section 5.1. Our comparison of different modeling languages is summarized in Table 6.1.

ESRA, ESSENCE and Zinc were developed to specify or model general combinatorial problems. Therefore, they do not provide any data structures or language elements occurring in staff scheduling problems, such as intervals, links, curves, derived properties and constraints. ESRA and ESSENCE provide only a finite set of predefined functions or constraints in order to model a problem, whereas Zinc allows to define additional functions and predicates to adapt and extend the language to a specific combinatorial optimization problem.

Some aspects of the ASPEN language are akin to TEMPLE, for instance, activities are similar to intervals and functional dependencies are comparable with derived properties. Furthermore, in ASPEN it is possible to model arbitrary functional dependencies by user-defined code in the programming language `C`, thus ASPEN is an open language in our sense. Since in the ASPEN language high-level goals are decomposed into low-level operations ASPEN is also a modular language. However, ASPEN is strongly focused on the characteristics of space craft operations and cannot be applied directly to staff scheduling problems.

Like TEMPLE, IBM ILOG OPL provides further language elements to facilitate the development of scheduling models such as intervals and cumulative functions which are similar to curves. However, in IBM ILOG OPL the user must use a fixed set of temporal or specialized constraints which cannot be extended further whereas in TEMPLE arbitrary constraints can be defined. Moreover, in TEMPLE, a user can select between different local search algorithms and he or she may influence the search process by specifying user defined moves. The IBM ILOG CPLEX CP Optimizer engine uses an exact method to obtain a solution for a considered scheduling task. A user may adjust some parameters of that method, but he cannot affect which regions of the search space shall be pruned or shall be explored at first hand.

As for Comet, TEMPLE inherited some of its syntax and data structures, e.g., sets, ranges and selectors, and Comet is the target language of our TEMPLE compiler. Some aspects of Comet's modular architecture, such as user-defined invariants, differentiable functions, and differentiable constraints, are similar to derived properties or constraints. However, the implementation of invariants, differentiable functions and constraints, is far more complex and time consuming in Comet, because entire classes have to be coded, whereas in TEMPLE properties and constraints can be usually defined within several lines of code. Comet is also an open language providing both predefined constraints as well as the possibility to define arbitrary user-defined properties or constraints of a problem. The main difference between TEMPLE and Comet is that TEMPLE offers abstractions and notations of staff scheduling problems, namely intervals, links and curves, reflecting common features of staff scheduling tasks. Moreover, we tried to design TEMPLE as simple as possible, thus, no knowledge about object orientation or any details on local search techniques, is required from a user.

6.2 Metaheuristic Frameworks

Beside modeling languages, metaheuristic frameworks represent a further possibility to develop algorithms for combinatorial optimization problems. Within a metaheuristic framework certain core functionalities of one or several metaheuristic techniques have already been realized. To develop an algorithm for a particular combinatorial optimization problem we have to extend the framework by providing problem specific information, e.g., a solution representation, moves, or an objective function.

In recent years, several metaheuristic frameworks have been proposed by different authors, e.g., OpenTS [30], EasyLocal++ [27], HOTFRAME [20], Templar [35] or ParadisEO [9]. We will shortly describe three frameworks which we looked at as possible candidates for the design of generic solutions for staff scheduling problems before we decided to develop a new domain specific language on our own.

6.2.1 OpenTS

OpenTS [30] is a Java-based framework to develop and implement Tabu Search algorithms [29]. To obtain a Tabu Search algorithm for a particular combinatorial optimization problem we must provide Java classes implementing the following items:

1. A *solution representation*.
2. An *objective function* evaluating solutions of the considered problem.
3. One or several *Moves* defining the local neighborhood computed for a considered problem.
4. If more than one kind of move should be used within the tabu search algorithm a user must also provide a so-called *move manager* class. Within that class the user must specify how different types of moves are chosen to compute the local neighborhood of a solution.

Optionally, we can further extend classes for the Tabu Search algorithm and the tabu list which are already existing within the framework, and we can specify additional aspiration criteria. In that way we can adapt the standard Tabu Search algorithm of OpenTS to a particular problem. Finally, we compile our classes containing problem specific information with the Java classes of the OpenTS framework and so we obtain an executable Tabu Search algorithm.

6.2.2 EasyLocal++

EasyLocal++ [27] is an object-oriented framework realized in the programming language C++. In contrast to OpenTS EasyLocal++ is not restricted to tabu search. EasyLocal++ provides several local search algorithms already existing within the framework, such as tabu search and simulated annealing, and it supports the development of further local search techniques. During the optimization process several local search techniques can be combined with each other, thus, in EasyLocal++ it is very easy to obtain hybrid metaheuristic algorithms.

According to Di Gaspero and Schaerf [27] the core of EasyLocal++ consists of a set of classes which are responsible for different aspects of a local search algorithm. These classes can be partitioned into the following five categories:

Data classes store the basic data of a local search algorithm, namely a state or solution in the search space, moves, input and out data.

Helpers are responsible for neighborhood computations, prohibition mechanisms, the computation and dynamic adaption of an algorithm's objective function, and for the creation of output data.

Runners execute runs of local search algorithms. They start at an initial solution, perform a series of moves and end up in a final search state.

Solvers create an initial solution and they control the entire search process by executing one or several runners.

Testers are used to debug algorithms, to tune parameters and to analyze local search algorithms.

To develop a local search algorithm in EasyLocal++ we must extend some classes and we must implement a few so-called *MustDef* methods. In these *MustDef* methods we specify how initial solutions are created, how the objective function value and hard constraint violations are obtained and how moves are computed. In addition, we can further adapt a local search algorithm to a particular optimization problem by overriding so-called *MayRedef* methods.

6.2.3 ParadisEO

ParadisEO [9], parallel and distributed evolving objects, is a metaheuristic framework supporting the design of both local search techniques and evolutionary algorithms. ParadisEO has the same architecture as EasyLocal++ consisting of helper, runner and solver classes:

- ▷ Helpers perform low-level actions related to the evolution or local search process, e.g., such as evolutionary operations or neighborhood exploration.
- ▷ Runners implement a certain metaheuristic technique themselves. They perform the run of an algorithm from an initial solution or population to the final one.
- ▷ Solvers are responsible for the control of the evolution process or the local search process, or a combination of both.

As in OpenTS and EasyLocal++, in ParadisEO we must provide at least information on an initial solution, objective functions, moves or genetic operators, to obtain an algorithm for a specific optimization problem, and by overriding already existing methods of the framework's classes, we can adapt an algorithm further to the specifics of a particular task.

In addition to OpenTS and EasyLocal++, in ParadisEO we can built hybrid algorithms consisting of evolutionary and local search components. In that manner we can combine the characteristics of both approaches within a single solution. Evolutionary algorithms are better in exploring the entire search space whereas local search techniques are more suited to intensify the search in particular regions.

Finally, ParadisEO enables the development of parallel and distributed solutions. Cost-intensive processes like neighborhood explorations or population evaluations can be distributed among several threads, multiple processor cores and different computers.

6.2.4 Comparison with TEMPLE

At the start of our research we considered metaheuristic frameworks as possible candidates for the design of generic solutions for staff scheduling problems. However, after it had become clearer to us, which design goals we wanted to achieve with our desired generic solution, we decided to develop a new domain specific language for staff scheduling problems on our own. Basically, we took that decision because of the two differences between metaheuristic frameworks and TEMPLE:

1. Metaheuritic frameworks are aimed at general combinatorial optimization problems. They do not provide abstractions and notations reflecting basic building blocks of staff scheduling problems, and thus, it is hard to realize the design goal modularity in metaheuristic frameworks.
2. When implementing an algorithm within a metaheurstic framework a user must possess knowledge of the architecture of the framework, object oriented programming, and sometimes even of local search techniques. These requirements on a potential user are in conflict with our design goal simplicity.

Chapter 7

The TEMPLE Compiler

We designed and implemented a TEMPLE compiler to transform TEMPLE programs into executable local search algorithms that solve the staff scheduling problems. As input the compiler is passed a problem model formulated in the TEMPLE modeling language and an XML-file containing input information of a particular problem instance. On the basis of that input the TEMPLE compiler generates three local search algorithms for the considered staff scheduling tasks: a simulated annealing algorithm [36], a hill climbing strategy [39] and an iterated local search algorithm [37]. The three local search algorithms are written in the constraint-based optimization language Comet [32]. To obtain a solution for the considered staff scheduling problem the generated algorithms are executed by the Comet optimization engine. Finally, the best solution found during the execution of a local search algorithm is returned as an XML-file. Figure 7.1 illustrates the entire approach we followed to solve staff scheduling problems in TEMPLE. In detail, the following files are generated by the TEMPLE compiler when transforming a TEMPLE model into classes of the Comet optimization language:

- ▷ For each *derived property* defined in the TEMPLE problem the TEMPLE compiler creates two files, a class file representing the derived property, and an invariant file. The class representing the derived property encapsulates the value it is responsible for evaluating moves, i.e., it computes the new value the property would take if a move was executed. On the other hand, the invariant file initializes the value of a property at the beginning of a local search algorithm and updates the property value whenever a move is actually performed.
- ▷ Also, for each *derived curve and constraint* the TEMPLE compiler creates a class file representing the derived curve or constraint and an invariant file. Again, the class file encapsulates a curve's state or a constraint's violation degree and evaluates changes resulting from moves, whereas the invariant file initializes and

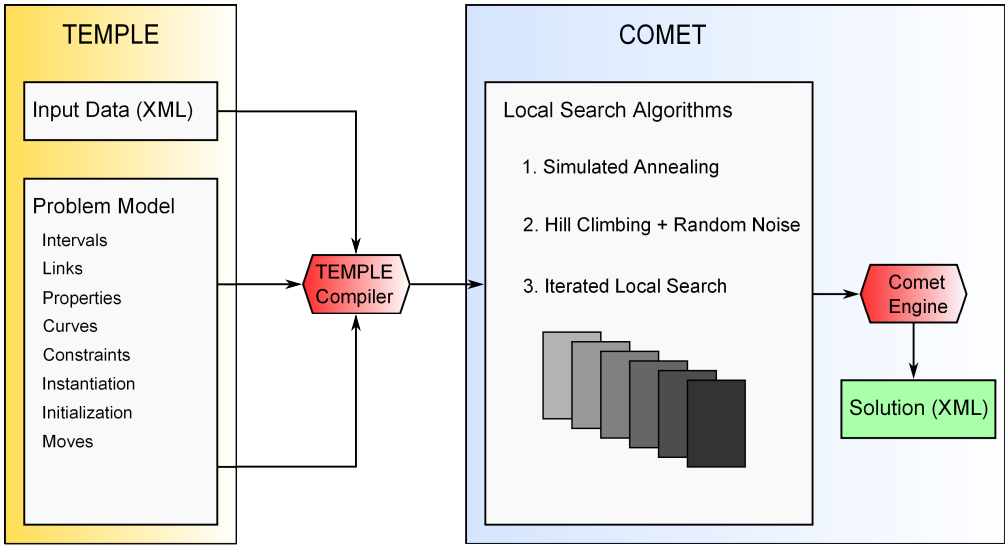


Figure 7.1: A Temple compiler transforms Temple models into three generic local search algorithms, that can be executed instantaneously.

updates a specific curve or constraint violation degree during a local search algorithm.

- ▷ For each *instantiation element* the TEMPLE compiler generates an instantiator class file to instantiate new additional intervals.
- ▷ For each *initialization element* the TEMPLE compiler builds an initializer class file which is used to compute initial values of basic interval properties, to restrict the domains for basic interval properties, and to link intervals with each other.
- ▷ For each defined *move* the TEMPLE compiler creates a corresponding Comet class, responsible for computing, evaluating and executing moves.
- ▷ For each *interval* declared in a TEMPLE model the TEMPLE compiler creates an interval class file. That class contains the basic properties associated with a specific interval and aggregates the derived properties, curves, constraints as well as the moves, instantiators and initializers defined for that interval.
- ▷ Finally, the TEMPLE compiler creates a class called `TimeIntervalModel`, which is the central management class of the compiled problem model. This class aggregates all created intervals, and administers two constraint systems for hard

and soft constraints. Each of the created local search algorithms interact only with time interval model during their search.

While implementing the TEMPLE compiler we had to solve several problems to ensure that the generated local search algorithms work correctly and efficiently. In the remainder of this chapter we will describe these tasks in more detail and we will show how we managed to solve them.

7.1 TEMPLE Model Analysis

As a first step, the TEMPLE compiler analyzes the structure of a TEMPLE program. Thereby, the compiler builds up a dependency graph storing the dependencies between basic properties, derived properties, curves, constraints, instantiation elements, and initialization elements. Figure 7.2 presents the dependency graph for the sample resource planning and staff scheduling problem in section 5.4. For instance, in Figure 7.2, the edge between a shift's derived properties `TotalBreakTime` and `TotalBreakTimeInPercent` indicates, that the latter property is derived from the former one. On the basis of a program's dependency graph the TEMPLE compiler performs the following two tasks:

1. The TEMPLE compiler detects superfluous elements, i.e., derived properties or curves which cannot be reached from a node representing a constraint, and excludes them from further processing steps.
2. The TEMPLE compiler detects directed circles within the dependency graph. Each derived element in a directed cycle depends transitively on itself, thus, the computation of such an element will not terminate. If a directed cycle is detected in a TEMPLE program, the compiler informs a user that the program cannot be processed correctly and terminates.

7.2 Computing an Initial Solution

7.2.1 Creating Intervals from the Input XML-File

As input each TEMPLE program is passed an XML-file which can be considered as a list of interval nodes, specifying the initial values and domains for each single interval as well as links between intervals. For instance, the following code listing presents an interval node of the input file for our sample resource planning and staff scheduling

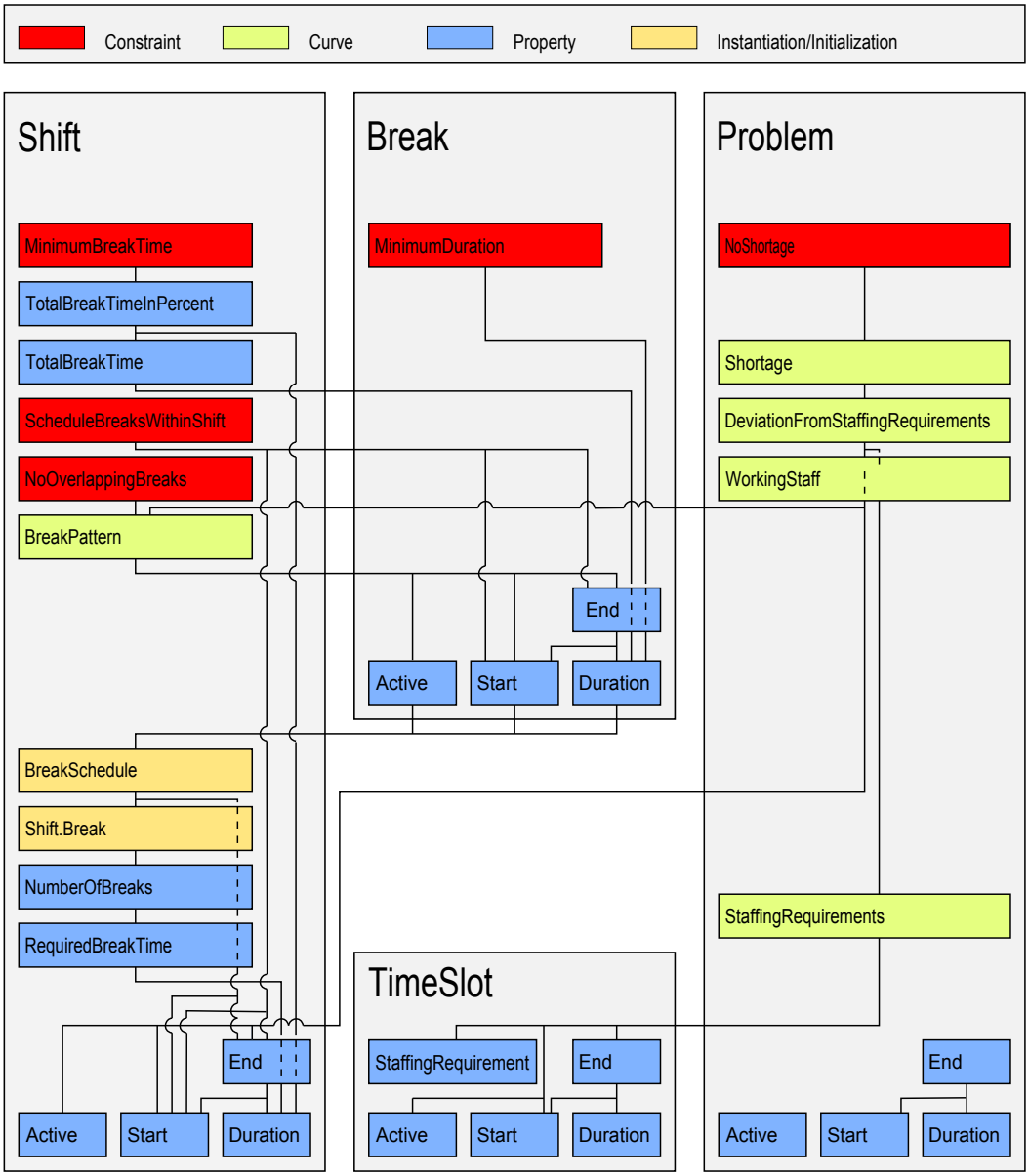


Figure 7.2: Dependencies existing between TEMPLE elements in the sample staff scheduling problem from Section 5.4.

problem. The node corresponds to the first shift of our sample problem. Initially, this shift is active, starts at 06:00, and lasts eight hours. The given domain values prohibit any changes of the basic decision variables by a local search algorithm:

```
<interval id="1001" type="Shift" description="Shift from 06:00 - 14:00">
   <basic-decision-variables>
      <variable-start>                        <!-- shift starts at 06:00 = time slot 36 -->
         <value>36</value>
         <domain>
            <domain-value>36</domain-value>   <!-- shift start must not be changed -->
         </domain>
      </variable-start>
      <variable-duration>                     <!-- shift lasts eight hours  = 48 time slots -->
         <value>48</value>
         <domain>
            <domain-value>48</domain-value>   <!-- shift duration must not be changed -->
         </domain>
      </variable-duration>
      <variable-activity>                     <!-- shift is active -->
         <value>1</value>
         <domain>
            <domain-value>1</domain-value>    <!-- shift may not be deactivated -->
         </domain>
      </variable-activity>
   </basic-decision-variables>
   <links>
      <!-- here the intervals linked to a shift could be specified -->
   </links>
</interval>
```

The information stored within the input XML-file is now transformed into interval objects as follows:

1. For each interval given in the input file a corresponding interval is instantiated.
2. The initial values specified within the XML-file are assigned to basic interval properties.
3. The domain values of basic properties are restricted to the domain values retrieved from the input file.
4. Intervals are linked with each other according to the information given within the input file.

After processing the data from the input XML-file we have created the first part of an initial solution for a particular problem instance. Figure 7.3 depicts the status of initialization after reading the input information for the sample staff scheduling problem from Section 5.4. Shifts, time slots and the root interval representing a problem have been instantiated and linked among each other. Moreover, the basic properties of the

created intervals, i.e., all elements drawn below the red line, have already been initialized. However, the bigger part of the initial solution, depicted with gray rectangles in Figure 7.3, is still not initialized at this stage.

Feasible Initialization Orderings

To initialize the remaining elements of a solution in correct order, we must consider the dependencies existing between them. In a feasible initialization ordering we initialize a single element only after we have already initialized all other elements it depends on. Within the TEMPLE compiler we obtain a feasible initialization ordering by traversing the dependency graph of an underlying staff scheduling problem in a depth-first-search order starting at the nodes representing constraints. Figure 7.4 shows a possible initialization ordering for the sample staff scheduling problem from Section 5.4.

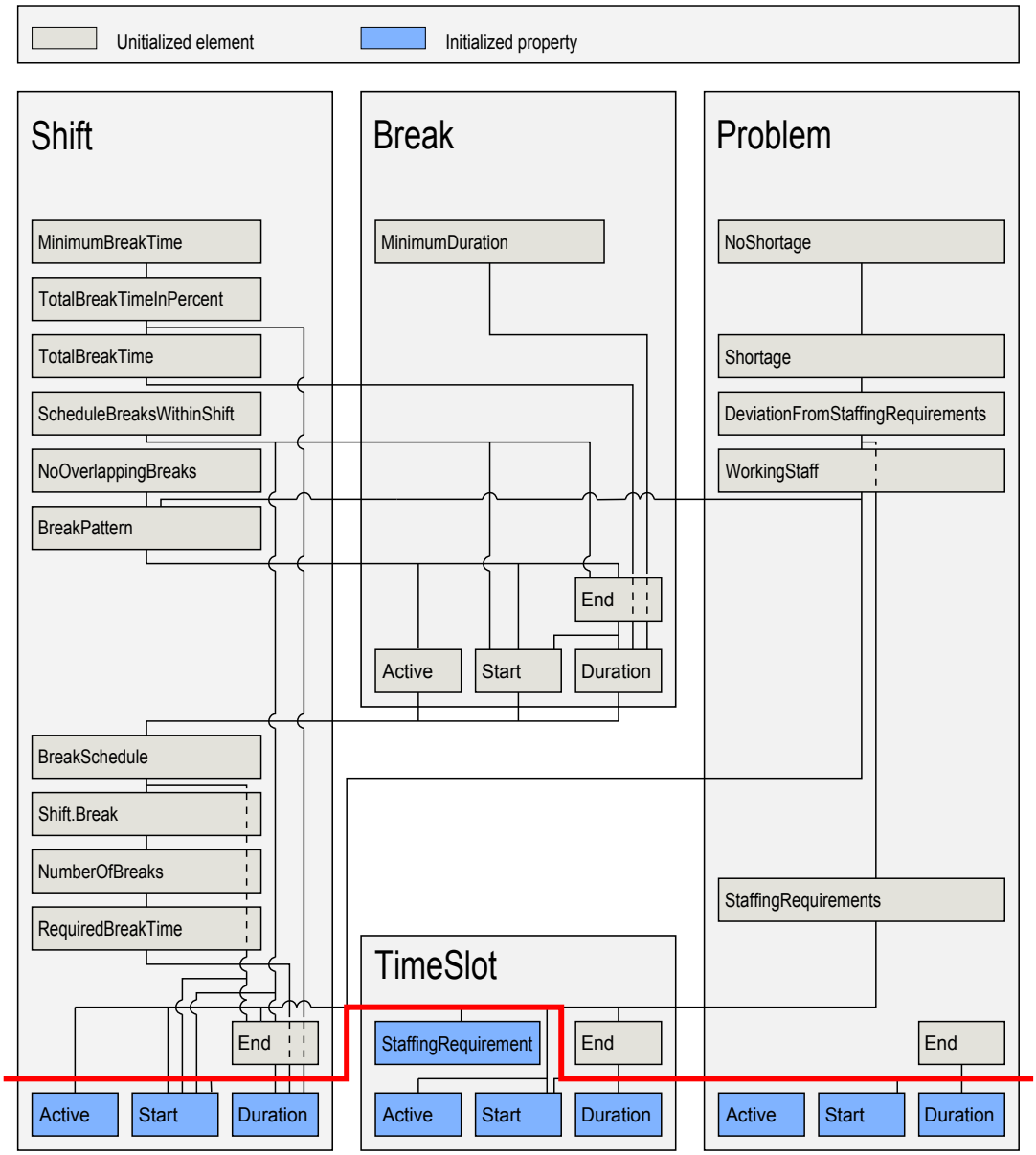


Figure 7.3: Initialized and uninitialized elements after processing the input data for our sample problem.

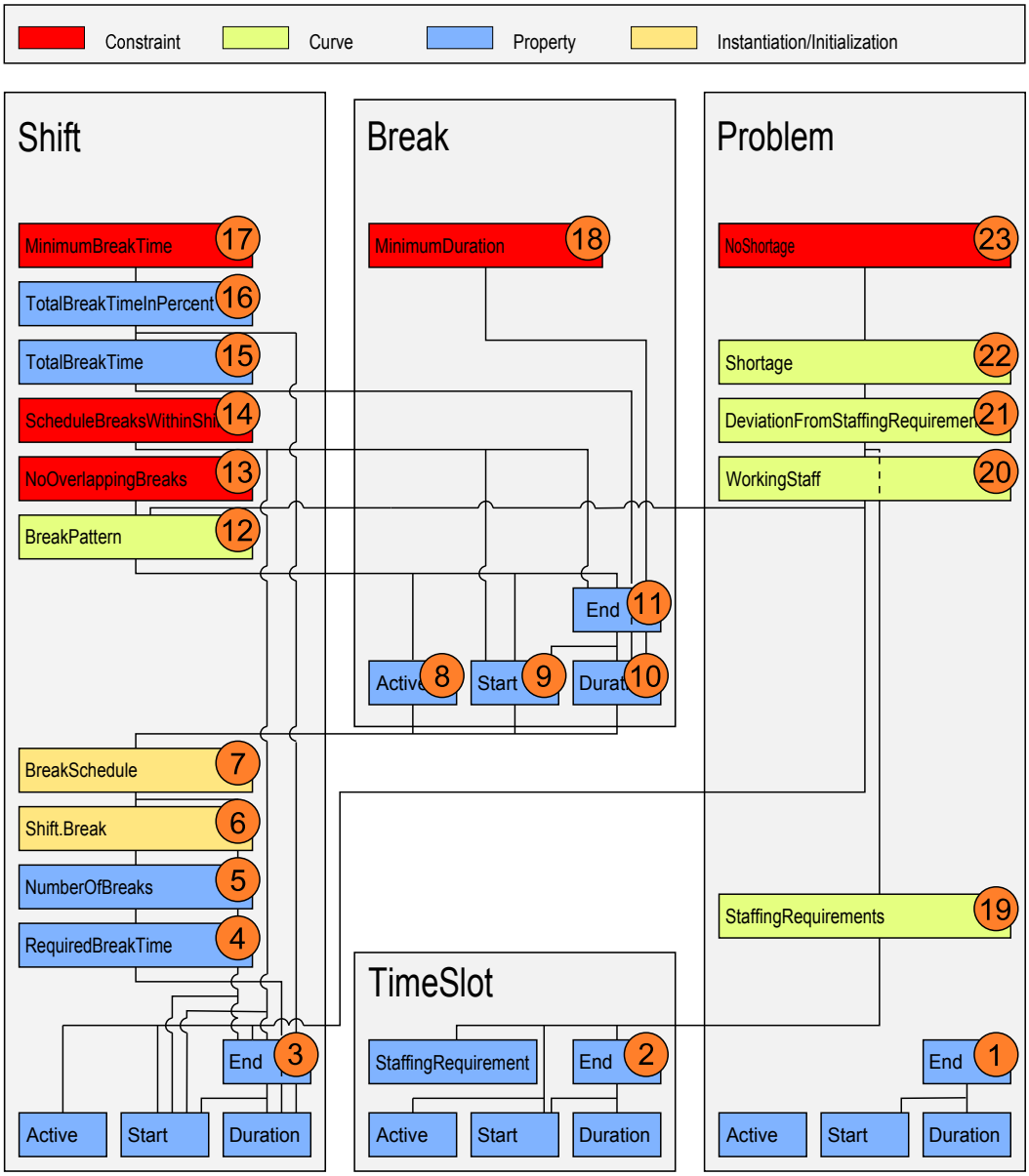


Figure 7.4: A feasible initialization ordering for the sample staff scheduling problem from Section 5.4.

7.2.2 Single Initialization Step

During the initialization of a of a derived property, curve, or constraint, the following two steps are performed:

1. We compute an initial property value, curve state, or constraint violation degree.
2. For each element we compute the basic decision variables which it depends on. Basic decision variables are used to check whether or not an element is affected by a move, and as we will see in Sections 7.4 and 7.5, they play a central role in the efficient evaluation and execution of moves.

The actual computation of initial property values, curves or constraint violation degrees takes place in the invariant classes created by our TEMPLE compiler. For each derived element the TEMPLE compiler analyzes the code snippet specifying how an element is derived. The compiler identifies the properties and curves which a derived element depends on and modifies the code snippet. The modified code snippet is inserted into an invariant's method `initPropagation` which initializes the derived element. For instance, the following code is inserted into an invariant to initialize a shift's derived curve `BreakPattern` in our sample staff scheduling problem from Section 5.4:

```
void InvariantCurveShiftBreakPattern::initPropagation()
{
   BreakPattern.Clear();

   forall ( i in scheduledBreak.getRange() )
      BreakPattern.Pulse( scheduledBreak[i].Start().value(),
                          scheduledBreak[i].End().value(),
                          scheduledBreak[i].Active().value() );
}
```

Basic decision variables are those variables representing basic interval properties `Start`, `Duration`, `Activity`. All derived elements depend either directly or transitively on basic properties. For a particular derived element we compute it's associated set of basic decision variable in the following manner:

1. If an element is a basic property, its associated set of basic decision variables contains only the decision variable representing the basic property.
2. Otherwise, for derived elements, we build the union of all sets of basic decision variables associated to the elements from which the considered element is derived. The resulting set is then associated to the considered element.

To compute the set of basic decision variables for each element, the TEMPLE compiler recognizes on which intervals, properties and curves an element depends on. In each class corresponding to a derived element the compiler creates a method called `Register` and inserts the code necessary to determine all basic decision variables. For a shift's derived curve `BreakPattern`, depending on the properties `Start`, `End`, and `Active` of the breaks scheduled in a shift, our TEMPLE compiler creates the following method:

```
void CurveShiftBreakPattern::RegisterVariables()
{
   forall(i in scheduledBreak.getRange())
      Register(scheduledBreak[i].Start()) ;

   forall(i in scheduledBreak.getRange())
      Register(scheduledBreak[i].End()) ;

   forall(i in scheduledBreak.getRange())
      Register(scheduledBreak[i].Active()) ;
}
```

In Figure 7.5 we see a single shift with four breaks and the corresponding initial values and sets of basic decision variables computed for basic break properties, the derived curve `BreakPattern`, and constraint `NoOverlappingBreaks`, as defined in our sample problem from Section 5.4. We see that for each element the set of basic decision variables is consists of all basic decision variables associated to its child nodes.

Figure 7.5: Initial values and sets of basic decision variables associated for basic and derived properties, curves and constraints of a shift and four breaks.

7.3 Move Computation

When specifying a move in TEMPLE, we consider the current solution of a particular optimization problem and derive a slightly changed new solution from it. More precisely we carry out the following steps:

1. We consider property values and curves of the current solution.
2. On the basis of the current property values and curve states we compute new values for basic interval properties, `Start`, `Duration`, `Active` or additional basic properties.
3. We assign the newly computed values to basic interval properties to obtain a new solution.

For instance, in move `PutBreakAtNewPosition` for our sample staff scheduling problem from Section 5.4, we consider a shift's `Start` and `End` in the current solution, we use these properties to compute a new break start within the shift, and we assign the new break start to a break's basic property `Start`:

```
Move Shift::PutBreakAtNewPosition(Shift thisShift, Shift.Break[] scheduledBreak)
{
  range S = thisShift.Start .. thisShift.End;

  select(i in scheduledBreak.getRange())
    select(newPosition in S)
       scheduledBreak[i].Start = newPosition;
}
```

Since a basic interval property is represented by a basic decision variable, a computed move consists of a set of basic decision variables and values that shall be assigned to those variables. To ensure that moves are represented and computed in exactly that manner our TEMPLE compiler analyzes the code specified for each move and detects assignment statements in which basic interval properties are involved. For each move our TEMPLE compiler generates a method called `Compute`, which stores pairs of basic decision variables and new values computed during its execution.

The following code snippet represents the method `Compute`, generated by the TEMPLE compiler, in order to compute the basic decision variables to be changed, and to determine values to be assigned by move `PutBreakAtNewPosition`:

```
void MoveShiftPutBreakAtNewPosition::Compute()
{
  range S = thisShift.Start().value() .. thisShift.End().value();

  select ( i in scheduledBreak.getRange() )
  {
    select ( newPosition in S )
    {
      this.StoreVariableValuePair( scheduledBreak[i].Start().value(), newPosition );
    }
  }
}
```

7.4 Move Evaluation

After computing a move we determine whether a move may be carried out at all, and we assess to what degree the quality of the current solution is improved or worsened by the move. To evaluate the impact of a move on the current solution, we carry out the following three steps:

1. We clarify if a move is *domain consistent*. For each basic decision variable of the move we check whether the value that shall be assigned to the variable is contained by the domain of the variable.
2. We determine whether or not a move violates any hard constraints of the considered optimization problem.
3. We compute the change in the soft constraint violation degree caused by the move.

If a move is domain consistent and satisfies all hard constraints, we call it a *feasible move*. Only feasible moves are allowed within the local search algorithm provided by TEMPLE to be carried out.

When executing a local search algorithm, the bigger part of running time is caused by evaluating moves. For that reason we take the following measures to reduce the effort for move evaluation and consequently increase the efficiency of local search:

- ▷ If a move is not domain consistent we stop its evaluation. Since the move is already classified to be infeasible and will not be carried out, we do not need to determine its effects on hard and soft constraints.
- ▷ When assessing a move's impact on hard constraints we apply lazy evaluation, i.e., if we observe that hard constraint is violated by the move we stop the further evaluation of other hard constraints and soft constraints.
- ▷ We use a move evaluation cache to avoid that a single move is evaluated several times for the same part of a problem's solution.
- ▷ When evaluating a move's effect on hard and soft constraints we only evaluate those parts of a solution that are really affected by the move. Basic properties, derived properties, curves and constraints are affected by a move if any of the basic decision variables changed by a move is contained in the set of basic decision variables associated with an element.

Figure 7.6 shows how the impact of a move is evaluated on an instance of hard constraint `NoOverlappingBreaks` in a solution to our sample problem from Section 5.4. Elements affected by the move are depicted in colors, unaffected elements are shaded gray.

The move assigns the first break in the third shift a new `Start`, namely `07:50`. Consequently, the move consists of a single basic decision variable, v_2, representing the break's start, and v_2 is assigned an integer value representing `07:50`.

To compute the new violation degree of constraint `NoOverlappingBreaks` we have to recompute all elements from which the constraint is derived, in our case this is curve `BreakPattern`. The curve itself depends on the properties `Start`, `End`, and `Active`, of each break scheduled within the shift. Since only the basic properties `Start` and `End` of one break depend on variable v_2, we have to recompute only the `Start` and `End` of that single break.

The re-computation of curve `NoOverlapping` reveals that if the move was executed two breaks would be overlapping and consequently the hard constraint would be violated. Thus, the considered move is an infeasible one.

To evaluate moves in that manner the TEMPLE compiler inserts methods called `GetNewValue`, `GetNewCurve`, and `GetNewViolationDegree` in every class representing a property, curve or constraint. When invoked these methods check whether a move has an effect on the corresponding element. If that is the case the method looks into the move evaluation cache to avoid multiple evaluations of the same element. Only if the result of the considered move is not cached, the value, curve or violation degree is recomputed. For that purpose the TEMPLE compiler inserts the user defined code used to derive an element. For each property or curve the element depends on, the TEMPLE compiler inserts `GetNewValue` or `GetNewCurve` methods to trigger the re-computation of these elements. The following code snippet shows the method `GetNewCurve` that is inserted into the class representing the derived curve `BreakPattern`.

```
Curve CurveShiftBreakPattern::GetNewCurve(Move moveToEvaluate)
{

  // 1. Check if move is affecting the Property
  if (!_basicVariables.HasCommonVariables(moveToEvaluate))
    return currentBreakPattern

  // 2. Check if move has already been evaluated in this iteration.
  if( this.IsCached(moveToEvaluate) )
    return this.GetCachedValue(moveToEvaluate);

  // 3. Recompute curve
  Curve newBreakPattern  = new Curve();

  forall ( i in scheduledBreak.getRange() )
  {
```

```
    newBreakPattern.Pulse( scheduledBreak[i].Start().GetNewValue(moveToEvaluate),
                           scheduledBreak[i].End().GetNewValue(moveToEvaluate),
                           scheduledBreak[i].Active().GetNewValue(moveToEvaluate) );
  }

  // Store recomputed curve in move evaluation cache
  this.Cache(moveToEvaluate, newBreakPattern);

  return newBreakPattern;
}
```

A single move *m* has the following three adjoint parameters: $m.IsFeasible$ specifies if a move may be applied to the current solution. $m.Fitness$ indicates the fitness of the solution obtained by applying a feasible move. *m.Delta* specifies the change of a solution's fitness if move *m* will be applied. *m.Delta* is positive if the quality of a solution is worsened whereas negative values for *m.Delta* show that the fitness of a solution will be improved when executing move *m*.

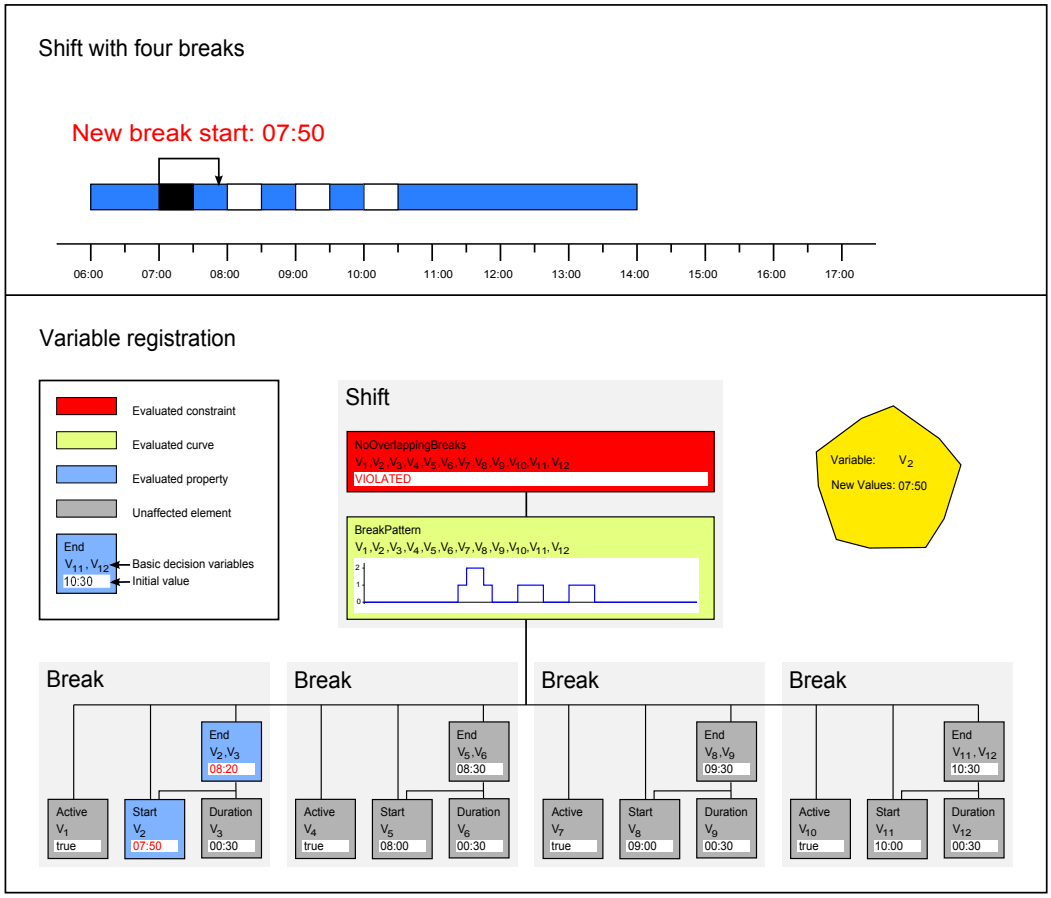


Figure 7.6: In the local search algorithms obtained with TEMPLE moves are evaluated only for those elements which they affect.

7.5 Move Execution

If a move is domain consistent and feasible it might be executed by a generic local search algorithm in order to obtain a new solution. The execution of a move is very simple. To each basic decision variable we assign the new value stored in the move.

Afterwards all elements hich depend on the basic decision variables must be updated in order to obtain the solution of the considered problem. Again, to ensure efficiency, we want only those parts of a solution to be updated that are actually affected by the move.

Figure 7.7 shows the effects of an executed move on parts of the solution for our sample resource planning and staff scheduling problem. The move places the fourth break break of a shift at `10:50`. All elements not affected by the move are shaded gray whereas those that are changed by the move are depicted with colored background. To execute the move the variable representing the break's start, v_{11}, is assigned the new start, `10:50`. Then all elements dependent on variable v_{11} are updated. We see that this move changes only as few derived elements as necessary: a single break's `End`, a single shift's `BreakPattern`. The updated hard constraint `NoOverlappingBreaks` indicates that the constraint is still satisfied after the move has been performed.

To ensure that a solution is really updated in that efficient manner we exploit features provided by Comet's user-definable invariants. In Comet's user-definable invariants we can specify from which source variables a certain target variable depends on. Whenever a source variable is changed by a move Comet automatically executes a property called `propagateInt` to update the values of the target variable.

In our case the target variable is a variable representing a derived property's value or a constraint's violation degree. For derived curves we introduced a trigger variable which may be used as target variable. For each derived element our TEMPLE compiler creates a user definable invariant and inserts code setting the source and target variables. For instance, for a shift's derived curve `BreakPattern`, the TEMPLE compiler creates the following code:

```
void ICurveShiftBreakPattern::post(InvariantPlanner<LS> planner)
{
  if (!_posted)
  {
    //source variables = properties on which BreakPattern depends on
    forall( i in scheduledBreak.getRange() )
      planner.addSource( scheduledBreak[i].Start().value() ) ;

    forall( i in scheduledBreak.getRange() )
      planner.addSource( scheduledBreak[i].End().value() ) ;

    forall( i in scheduledBreak.getRange() )
      planner.addSource( scheduledBreak[i].Active().value() ) ;
```

```
    //target variable = trigger variable of BreakPattern
    planner.addTarget(BreakPattern.trigger());

    _posted = true;
  }
}
```

In addition our TEMPLE compiler also generates the code for method `propagateInt`, which will be used to update the values, curves and violations degrees, to obtain a new solution. For the derived curve `BreakPattern` method `propagateInt` looks as follows:

```
void ICurveShiftBreakPattern::propagateInt (boolean notLastInvokation, var{int} variable)
{
  bool isLastInvokation = ! notLastInvokation;

  //update curve only once, after all source variables have been changed
  if(isLastInvokation)
  {
    BreakPattern.Clear();

    forall ( i in scheduledBreak.getRange() )
      BreakPattern.Pulse( scheduledBreak[i].Start().value(),
                          scheduledBreak[i].End().value(),
                          scheduledBreak[i].Active().value() );

    //change the value of trigger variable
    BreakPattern.pullTrigger();
  }
}
```

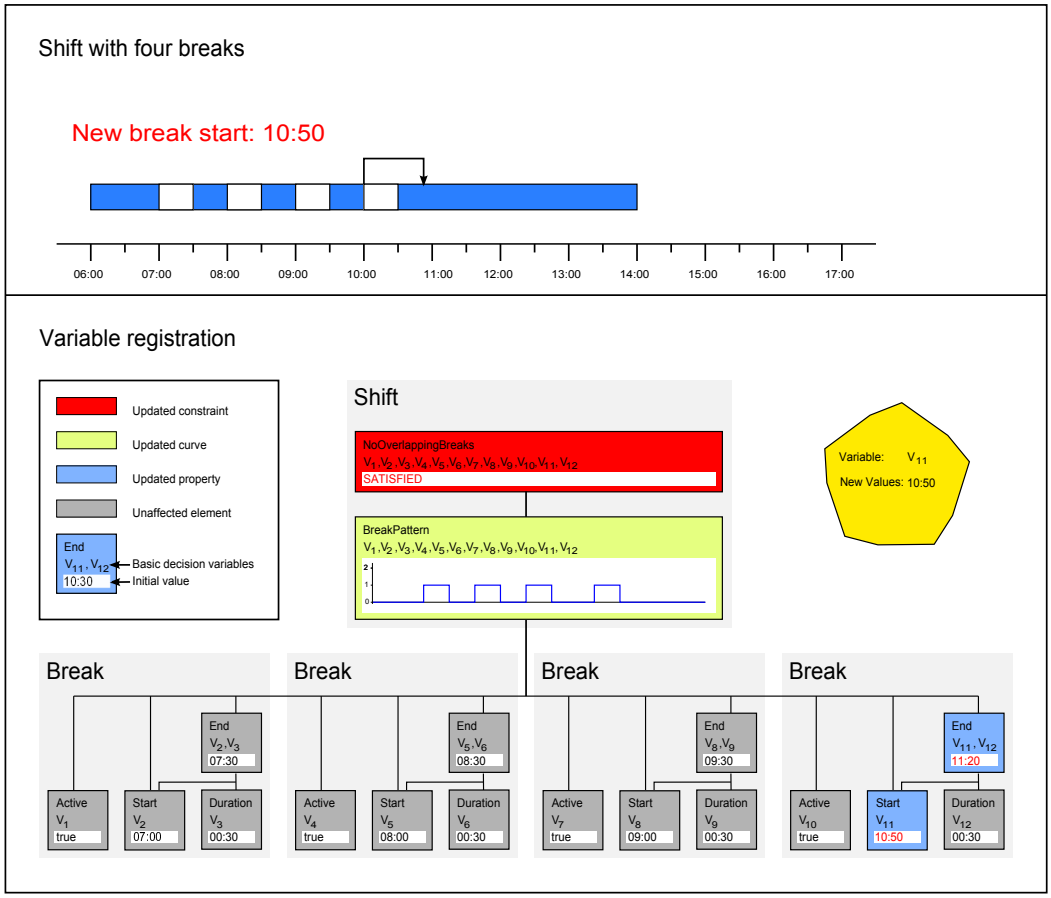


Figure 7.7: In the local search algorithms obtained with TEMPLE only those parts of a solution are changed which are actually affected by the move.

7.6 Efficient Curve Evaluation

7.6.1 Motivation

The costs for evaluating a move's effect on derived curves are relatively high compared to the effort for evaluating moves on derived properties or constraints. For instance, let us consider the derived curve `WorkingTime` from Section 5.3.5, representing the times when an employee is actually working and not having a break. This curve may be derived in three steps as follows:

1. For a shift we derive a curve called `AttendanceTime`, which is set to one along the duration of a single shift.

```
Curve Shift::AttendanceTime(Shift thisShift)
{
  AttendanceTime.Pulse ( thisShift.Start,
                         thisShift.End,
                         thisShift.Active );
}
```

2. We introduce a curve representing a shift's `BreakPattern`, having a value of one whenever a break occurs.

```
Curve Shift::BreakPattern(Shift.Break[] scheduledBreak)
{
  forall(i in scheduledBreak.getRange())
  {
    BreakPattern.Pulse( scheduledBreak[i].Start,
                        scheduledBreak[i].End,
                        scheduledBreak[i].Active );

  }
}
```

3. Finally, we derive the curve representing a single employee's `WorkingTime` by subtracting curve `BreakPattern` from curve `AttendanceTime`.

```
Curve Shift::WorkingTime(Shift thisShift)
{
  WorkingTime.Add      ( thisShift.AttendanceTime );
  WorkingTime.Subtract ( thisShift.BreakPattern   );
}
```

Figure 7.8 shows a single shift having two breaks and depicts how the curves `AttendenceTime`, `BreakPattern` and `WorkingTime` are derived from that shift. To illustrate the great computational costs associated with derived curves let us consider the effects of a simple move on these three derived curves. Figure 7.9 highlights the positions which are changed if we schedule the one-hour break starting at `11:00` at `13:00`.

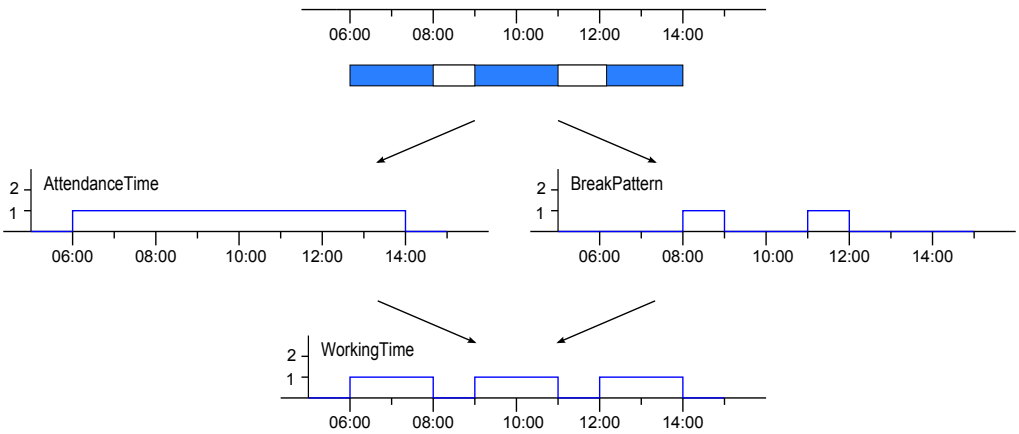


Figure 7.8: Derived curves `AttendenceTime`, `BreakPattern` and `WorkingTime` resulting from a single shift having two breaks.

We see that only two time ranges from `11:00` to `12:00` and from `13:00` to `14:00` are actually changed in curve `BreakPattern` and `WorkingTime`. However, under the assumption that our planning period consists of ten minute time slots, the evaluation of that simple move would require 108 arithmetic operations with the code generated by our TEMPLE compiler:

	No. of Operations	
`AttendanceTime`	0	The curve is not affected by the move.
`BreakPattern`	12	`Pulse` is called for both one hour breaks.
`WorkingTime`	96	`BreakPattern` is subtracted from `AttendenceTime`.
Total	108	

The reasons for these great computational costs are twofold:

1. We recompute curves completely from the scratch.
2. We do not record, propagate and exploit the information on which positions have been changed to what degree.

Therefore, we will develop a speed-up strategy and an appropriate data structure in order to accelerate the evaluation of derived curves significantly.

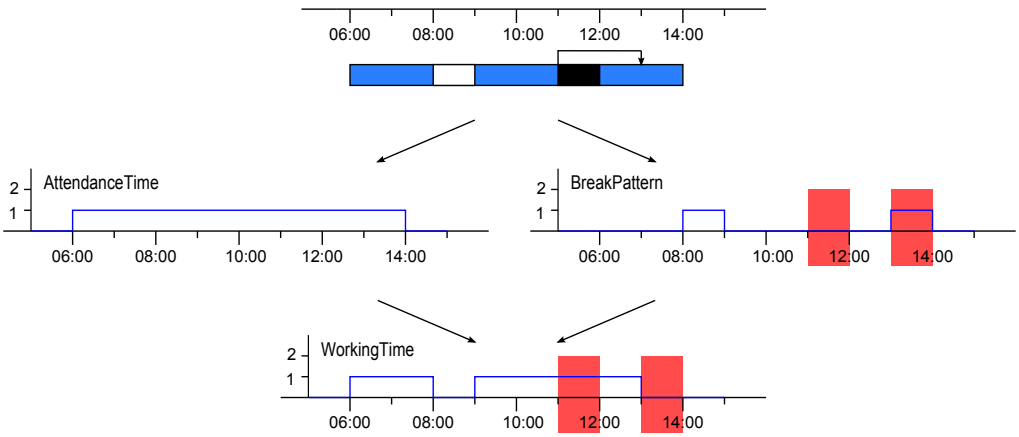


Figure 7.9: Changed positions (red shaded areas) in derived curves caused by a single move.

7.6.2 A Speed-Up Strategy

Figure 7.10 (a) shows how the effects of a move on a particular curve are evaluated so far. We consider an empty curve $Curve_0$, containing exclusively zeros at each position, carry out several operations o_1, o_2 and o_3, and obtain the curve resulting from that particular move $Curve_{move}$.

$$Curve_{move} = Curve_0 \circ o_1 \circ o_2 \circ o_3$$

Figure 7.10 (b) depicts an alternative way to compute $Curve_{move}$. Starting from a curve's current state in the current solution $Curve_{current}$ we apply a several operations u_1, u_2 and u_3 to transform $Curve_{current}$ into $Curve_0$. Afterwards we execute again the operations o_1, o_2 and o_3 and obtain $Curve_{move}$ as our final result:

$$Curve_{move} = \underbrace{Curve_{current} \circ u_1 \circ u_2 \circ u_3}_{=Curve_0} \circ o_1 \circ o_2 \circ \circ o_3$$

Obviously, the computational costs associated with that alternative approach are much higher. However, under the assumption that the compositions of curve operations is commutative and associative, we can simplify the computation in the following manner:

1. We change the order in which the operations are carried out.

$$Curve_{move} = Curve_{current} \circ u_1 \circ o_1 \circ u_2 \circ o_2 \circ u_3 \circ o_3$$

2. We substitute each operation pair $u_i \circ o_i$ by two other methods $u'_i \circ o'_i$ having the same combined effect, but lower computational costs, $costs(u'_j \circ o'_j) < costs(o_j) < costs(u_j \circ o_j)$.

$$Curve_{move} = Curve_{current} \circ \underbrace{(u_1 \circ o_1)}_{=\circ u'_1 \circ o'_1} \circ \underbrace{(u_2 \circ o_2)}_{=u'_2 \circ o'_2} \circ \underbrace{(u_3 \circ o_3)}_{=u'_3 \circ o'_3}$$

$$Curve_{move} = Curve_{current} \circ (u'_1 \circ o'_1) \circ (u'_2 \circ o'_2) \circ (u'_3 \circ o'_3)$$

3. Finally, we again change the execution order of operations.

$$Curve_{move} = Curve_{current} \circ u'_1 \circ u'_2 \circ u'_3 \circ o'_1 \circ o'_2 \circ o'_3$$

The simplified approach is shown in Figure 7.10 (c). We see that the simplified computation takes less running time than the re-computation of a curve from the scratch. Moreover, Figure 7.10 (d) shows that it even suffices to substitute only some method pairs if the performance gain resulting from each single substitution is high enough.

7.6.3 Implementing the Speed-Up Strategy

To exploit the speed-up strategy for our purposes we developed a new class called `DeltaCurve` and use it in each `GetNewCurve` method to evaluate a move's effect on curves. In short class `DeltaCurve` may be characterized as follows:

- ▷ `DeltaCurve` provides the same operations as ordinary curves which are shown in Table 5.1.
- ▷ Additionally, for each ordinary operation `DeltaCurve` provides an undo operation, reverting the effects of its counterpart.
- ▷ `DeltaCurve` wraps the state of a curve in the current solution. Instead of changing the current state of the curve itself `DeltaCurve` stores the positions that are changed and records the difference, or delta, between the changed value and the original one in a curve's current state.

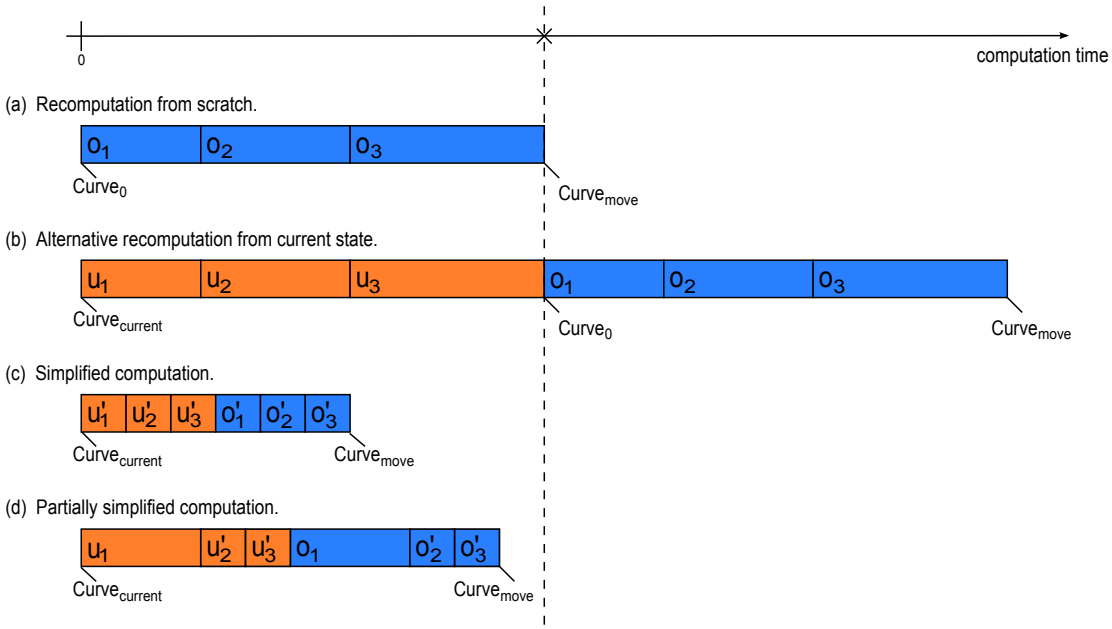


Figure 7.10: Simplifying the evaluation of curves.

By the help of class `DeltaCurve` we can now implement our alternative approach shown in Figure 7.10 (b). In each `GetNewCurve` method, we use an instance of `DeltaCurve` to transform a curve from its current state into the curve resulting from a particular move. Basically, in each `GetNewCurve` method we perform the following three tasks:

1. We create a new instance of `DeltaCurve` wrapping a curve's state in the current solution.

2. We apply undo operations to revert the effects resulting from the current solution. The arguments passed to undo operations are computed from the property values and curves within the current solution. After this step the applied `DeltaCurve` instance corresponds to a curve having only zero values at each position.

3. We obtain the curve resulting from a particular move by applying ordinary operations. The arguments passed to these operations are computed from the new property values and new (delta) curves resulting from the considered move. Finally, the `DeltaCurve` instance represents the curve resulting from that move.

The following code snippet shows an implementation of method `GetNewCurve` for the derived curve `BreakPattern` from section 7.6. Within this method we use undo operations as well as ordinary operations to evaluate the changes resulting from a move. We consider the move from Figure 7.8 shifting a break from `11:00` to `13:00` and in comments we report the changed positions and value differences recorded by the `DeltaCurve` instance. The applied undo operations remove the breaks from their positions in the current solution whereas original operations propagate the new break positions resulting from the move.

```
DeltaCurve CurveShiftBreakPattern::GetNewCurve(Move moveToEvaluate)
{
  //1. DeltaCurve wraps state of the curve in the current solution.
  DeltaCurve deltaBreakPattern = new DeltaCurve(currentBreakPattern);

  //2. Undo operations revert the effects resulting from current values.
  forall(i in scheduledBreak.getRange())
  {
    //breaks at current positions are removed from curve
    deltaBreakPattern.UndoPulse( scheduledBreak[i].Start().value(),
                                 scheduledBreak[i].End().value(),
                                 scheduledBreak[i].Active().value() );
  }
  //State of deltaBreakPattern.
  //changed positions: [08:00, 09:00], [11:00, 12:00]
  //delta:                          -1,             -1
  //no. of operations:  12

  //3. Original operations propagate effects resulting from new values.
  forall(i in scheduledBreak.getRange())
  {
    //breaks at new positions are added to curve
    deltaBreakPattern.Pulse( scheduledBreak[i].Start().GetNewValue(moveToEvaluate),
                             scheduledBreak[i].End().GetNewValue(moveToEvaluate),
                             scheduledBreak[i].Active().GetNewValue(moveToEvaluate) );
  }
  //State of deltaBreakPattern.
  //changed positions: [08:00, 09:00], [11:00, 12:00], [13:00, 14:00]
  //delta:                           0,             -1,             +1
  //no. of operations:  12

  deltaBreakPattern.RemoveUnchangedPositions();
  //State of deltaBreakPattern.
  //changed positions: [11:00, 12:00], [13:00, 14:00]
  //delta:                          -1,             +1

  return deltaBreakPattern;
}
```

We see that an instance of class `DeltaCurve` records the changed positions and the value differences caused by a move. Now, we will exploit that information to simplify the following operations and their corresponding undo operations: `Add`, `Subtract`, `CyclicAdd`, `CyclicSubtract`, `AddPositiveValues`,

AddNegativeValues, SubtractPositiveValues, and SubtractNegativeValues. These operations add or subtract other, already computed delta curves to or from the delta curve to be obtained. The computed delta curve contains the positions changed by the move and the value differences at these positions, thus, ordinary operations must only propagate these changes. The corresponding undo operations simply do not have to do anything at all. Simplified undo operations and simplified original operations have the same combined effect as their unsimplified counterparts. The following code sample shows method GetNewCurve of the derived curve WorkingTime applying simplified undo operations and ordinary operations:

```
DeltaCurve CurveShiftWorkingTime::GetNewCurve(Move moveToEvaluate)
{
  //1. DeltaCurve wraps state of the curve in the current solution.
  DeltaCurve deltaWorkingTime = new DeltaCurve(currentWorkingTime);

  //2. Simplified undo operations revert the effects resulting from current curves.
  deltaWorkingTime.UndoAdd     ( thisShift.AbsenceTime().Curve()  );
  deltaWorkingTime.UndoSubtract( thisShift.BreakPattern().Curve() );

  //UndoAdd and UndoSubtract do not do anything.

  //State of deltaWorkingTime.
  //changed positions:  -
  //delta:              -
  //no. of operations:  0

  //3. Simplified ordinary operations propagate effects resulting from new values.
  deltaWorkingTime.Add     ( thisShift.AbsenceTime().GetNewCurve(moveToEvaluate) );
  deltaWorkingTime.Subtract( thisShift.BreakPattern().GetNewCurve(moveToEvaluate));

  //AbsenceTime remains unchanged, thus Add does not propagate any changes.
  //Subtract propagates the changes in BreakPattern

  //State of deltaWorkingTime.
  //changed positions: [11:00, 12:00], [13:00, 14:00]
  //delta:                          +1,             -1
  //no. of operations:  12

  deltaWorkingTime.RemoveUnchangedPositions();

  return deltaWorkingTime;
}
```

Simplified undo operations do not perform any actions at all and consequently they do not cause any computational costs. Simplified ordinary operations only propagate the changes resulting from the evaluation of curve AbsenceTime and BreakPattern. Since AbsenceTime is not changed at all no changes have to be propagated by method Add. Method Subtract propagates only those changes that took place in curve BreakPattern at the positions between [11:00, 12:00] and [13:00, 14:00]. Due to our speed-up strategy the evaluation of that curve needs now only 36 arithmetic operations:

	No. of Operations	
`AttendanceTime`	0	The curve is not affected by the move.
`BreakPattern`	24	`UndoPulse` and `Pulse` are called for both one hour breaks.
`WorkingTime`	12	Changes in `BreakPattern` are propagated to `WorkingTime`.
Total	36	

The previous example illustrated that our modifications might indeed achieve a notable performance gain. In real-life staff scheduling problems we usually define curves modeling staffing requirements or available staff for entire weeks, months or years. The costs associated with the re-computation of such curves are many times higher than in our small example and, consequently, also the performance gain achieved by our speed-up strategy is significantly bigger.

7.6.4 A Note on the Correctness of the Speed-Up Strategy

Finally, we want to argue that the modified `GetNewCurve` methods, implementing our speed-up strategy, evaluate a move's effect on curves correctly. To do so, we introduce several restrictions on the code in method `GetNewCurve` and also for the code within the definition of derived curves. Under these restrictions we can guarantee the correctness of our speed-up strategy:

1. The code must not contain any if-statements or any statements using if-statements internally, e.g., min or max index selectors.
2. The code must not contain any loops except forall-loops.
3. Forall-loops must iterate over constant index ranges, i.e., index ranges do not depend on basic interval properties, derived properties or derived curves.
4. The indices used to access intervals must be constant, i.e., they must not be computed from basic interval properties, derived properties or derived curves.

Under these restrictions the same number of undo operations and ordinary operations are carried out during a single invocation of a method `GetNewCurve`. Moreover, the i-th undo operation corresponds to the i-th ordinary operation. Each undo and ordinary operation can be represented as a set of arithmetic operations $< p, +, v >$, increasing the value of a curve at position p by the value v. Since the addition of real or integer numbers is an associative and commutative operation, the composition of undo and ordinary operation is also associative and commutative. Thus, we may change the order in which operations are carried out, we may substitute undo and ordinary operations by

simplified operations, and we may reorder them again, as we do it in our speed-up strategy proposed in Section 7.6.2. Consequently, with the proposed speed-up strategy the evaluation of a move's effect on a curve will be carried out correctly, if the definition of a curve satisfies the previously stated restrictions.

7.7 Adaptive Local Neighborhood Computation

The generic local search algorithms generated by the TEMPLE compiler compute a local neighborhood of a current solution in the same manner. Internally, they represent a local neighborhood as a set of moves, and these moves are computed from the input specified by a user. In each iteration of a local search algorithm we have to decide which concrete moves should enter a local neighborhood. One possible approach might be to treat all moves equally, such that all moves would enter the local neighborhood with the same likeliness. However, there are several reasons arguing against such a policy:

- ▷ Some moves may perform better than others. Even a very experienced user can only assess roughly how good a move will behave in practice. Thus, the moves in a single TEMPLE program represent only a user's believe on how a solution could be improved. Consequently, in each TEMPLE program there are some moves improving solutions very strongly, other moves will improve solutions only marginally, and even some moves might not be able to improve a solution at all.
- ▷ The impact of a single move can vary over time. For instance, in our sample staff scheduling problem in Section 5.4 we introduced a single move, repositioning only a single break in each step. This move works well for our small example. For larger problem instances it also performs very well at an early stage, when the solution is of quite poor quality. At a later point of time this move becomes less effective because it considers only single shifts and breaks. On the other hand, moves considering the global state of the current solution, changing several breaks in several shifts in accordance with each other, are more likely to improve an already optimized solution at a later point of time.
- ▷ Different moves are differently expensive in terms of computational cost. When assessing the performance of a move we must consider both the improvement of the objective function value resulting from a move as well as the time needed to compute and evaluate a move.

Instead of selecting each move with the same likelihood at every point of time we associate a selection probability that shall be adapted during the execution of a local search algorithm. When computing a move's selection probability we must satisfy the following requirements:

1. Moves that caused more and bigger improvements should be selected with high probability. Moves that did not perform so well shall be selected with low probability.

2. A move's selection probability shall be recomputed in each iteration of the local search algorithm. Moreover it shall be based only on the recent history of the local search algorithm, e.g., for the last 1,000 iterations. In that way the selection probability of a move is dynamically adapted to the recent performance of a move.

3. The costs for computing and evaluating moves shall also be reflected by a move's selection probability.

4. A move's selection probability shall be strictly greater than zero. Each move must be given a chance to increase its associated selection probability. In that way we prohibit that a single move is excluded completely from the local search algorithm.

In order to compute and adapt a move's selection probability we introduce a new measure called *average improvement per second*, reflecting the improvement of the objective function caused by a move as well as the computational costs associated with a move. To compute the improvement per second each move has an attached history of limited size, in our implementation that size is 1,000 iterations.

After a move *m* is computed and evaluated we store the resulting improvement within its associated history. If the move is infeasible or worsens the quality of the current solution the stored improvement will be zero. Moreover we record the computation time in seconds needed to compute and evaluate the move. The average improvement per second is derived from a move's history by dividing the summed improvements of objective function values by the summed computation times.

The average improvement per second is used to determine the selection probability for each move. Each selection probability consists of a constant fraction and a dynamic fraction. In our implementation, the constant fraction is obtained by dividing 30% of the overall selection probability by the number of moves. In that way, we ensure that even very badly performing moves can enter a local neighborhood and they have a chance to increase their selection probability again. The dynamic fraction is obtained by distributing the remaining 70% proportionally among the moves according to their associated average improvement per second. Consequently, moves that have performed better in the past are more likely to enter the adaptive local neighborhood. The moves that enter a local neighborhood at a certain iteration are chosen by roulette wheel selection in accordance to their selection probability.

The obtained adaptive local neighborhood satisfies the four requirements we have previously stated and is used by three generic local search algorithms provided by TEMPLE.

Algorithm	*Parameter*	*Description*
Hill Climbing	t	Run time limit in seconds.
	p_{noise}	Probability for introducing random noise.
	s	Size of the local neighborhood computed in each iteration of the hill climbing algorithm.
Iterated Local Search	t	Run time limit in seconds.
	H	The hill climbing component ends after H iterations without any improvement of the objective value.
	P	The pertubation phase ends after P moves worsening the objective function values have been performed.
	s	Size of the local neighborhood computed in each iteration of the hill climbing algorithm.
Simulated Annealing	t	Run time limit in seconds.
	p_{init}	Selection probability of a worse solution with average fitness loss at the beginning of the simulated algorithm.
	p_{final}	Selection probability of a worse solution with average fitness loss at the end of the simulated annealing algorithm.

Table 7.1: Control parameters of TEMPLE's generic local search algorithms.

7.8 Control Parameters of the Generic Local Search Algorithms

As mentioned in the previous sections each of the three generic local search algorithms generated by our TEMPLE compiler is controlled by several parameters. Table 7.1 presents a short description of parameters involved in each single algorithm.

In TEMPLE, we can specify parameter values explicitly within a TEMPLE program. The following lines of code indicate how the parameter values are chosen for each single local search algorithm:

```
//hill climbing with random noise
algorithm running time  = 5 minutes;
algorithm               = hill climbing(0.05, 10);

//iterated local search
algorithm running time  = 5 minutes;
algorithm               = iterated local search(1000, 5, 10);

//simulated annealing
algorithm running time  = 5 minutes;
algorithm               = simulated annealing(0.5, 0.01);
```

7.9 Solving Staff Scheduling Problems

Besides many other files, our TEMPLE compiler creates a file called `Run.co` which contains all necessary information for optimizing a considered problem instance: the XML-file containing the input data of a considered staff scheduling problem instance, the local search algorithm that shall be applied to that instance, the control parameter values chosen for that algorithm, and an output XML-file where the obtained solution shall be stored. Thus, to apply a local search algorithm to a considered problem instance we simply have to execute the file `Run.co` on the Comet optimization engine within the Windows command prompt:

```
C:\Dynadec\Comet 2.10\compiler\comet.exe Run.co
```

Chapter 8

Practical Applications

In Chapter 5 we considered small, comprehensible sample staff scheduling problems to introduce and illustrate the basic abstractions, notations and concepts of TEMPLE. However, we developed TEMPLE to model and solve arbitrary real-life resource planning and staff scheduling problems. In this chapter we consider two complex real-life staff scheduling problems and show that they can be both effectively modeled as well as efficiently solved with TEMPLE.

First of all, we reconsider the break scheduling problem for supervisory personnel from Chapter 3 and we sketch how we managed to model it with TEMPLE. We compare the features of our TEMPLE model with those of the algorithm presented in Chapter 3 and observe that in TEMPLE we can derive a very small, concise and modular program for the considered break scheduling task. Finally, we evaluate the iterated local search algorithm obtained with TEMPLE on benchmark instances and compare the results obtained with TEMPLE to those reported in Chapter 3. For the considered benchmark instances the generic local search algorithm is able to return solutions of good quality in acceptable time.

Secondly, we will consider a further real-life break staff scheduling problem in which, like in the break scheduling problem for supervisory personnel, we have to compute a legal break schedule for the deployed personnel. In addition, we must also assign tasks to available employees in accordance with their qualifications. To tackle this complex problem we decompose it into three phases: In the first phase we compute a legal break schedule, in the second phase we optimize the break schedule with respect to task requirements, and in the final phase we determine a suitable task assignment. Each of these phases is modeled and solved by a separate TEMPLE program. The TEMPLE programs represent the core of a commercial break scheduling and task assignment tool which we developed for a customer of the consulting company Ximes Corp. We will report how that software is used at the customer's site in practice and we will demonstrate

that our approach is able to create high quality solutions within reasonable time.

8.1 A TEMPLE Model of the Break Scheduling Problem for Supervisory Personnel

The break scheduling problem for supervisory personnel could be modeled very easily in TEMPLE. Nevertheless, at this point we spare the reader the entire program, instead we illustrate the TEMPLE model in two structograms, shown in Figure 8.1 and Figure 8.2. These structograms depict each TEMPLE element involved in the model and show how the single elements are derived from each other. Each single element is associated a number specifying the lines of code which were needed to formulate the element. The complete TEMPLE program is presented in Appendix A of this thesis.

Constraints

In our model the five criteria concerning the legality of the break pattern, constraints C_1 *Break Positions* through C_5 *Break Durations*, are formulated as hard constraints, whereas shortage and excess of staff, constraints C_6 *Shortage of Employees* and C_7 *Excess of Employees* are modeled as soft constraints. In addition, we introduced two additional hard constraints, *NoOverlappingBreaks* and *ScheduleBreaksWithinShift*, to ensure that breaks are scheduled completely within the range of their shifts and to guarantee that breaks do not overlap each other. The concrete constraint formulations are presented in the appendix of this thesis, A.3 - A.10.

Figure 8.1 visualizes how we modeled each of these constraints. For instance, let us consider the two soft constraints C_6 *Shortage of Employees* and C_7 *Excess of Employees*, requiring that shortage and excess of staff shall be reduced to a minimum degree within a good solution. For each employee we introduce a curve representing the actual working time. By summing up all these single curves we obtain a curve modeling the available staff. In the next step we subtract the staffing requirements from the available staff and so we obtain the deviations from staffing requirements. Then we extract the negative deviations to obtain a curve representing the shortage of staff, determine the total amount of shortage associated with a solution, and finally we impose a constraint requiring that shortage of employees should be avoided. The constraint penalizing excess of employees is derived in a similar manner.

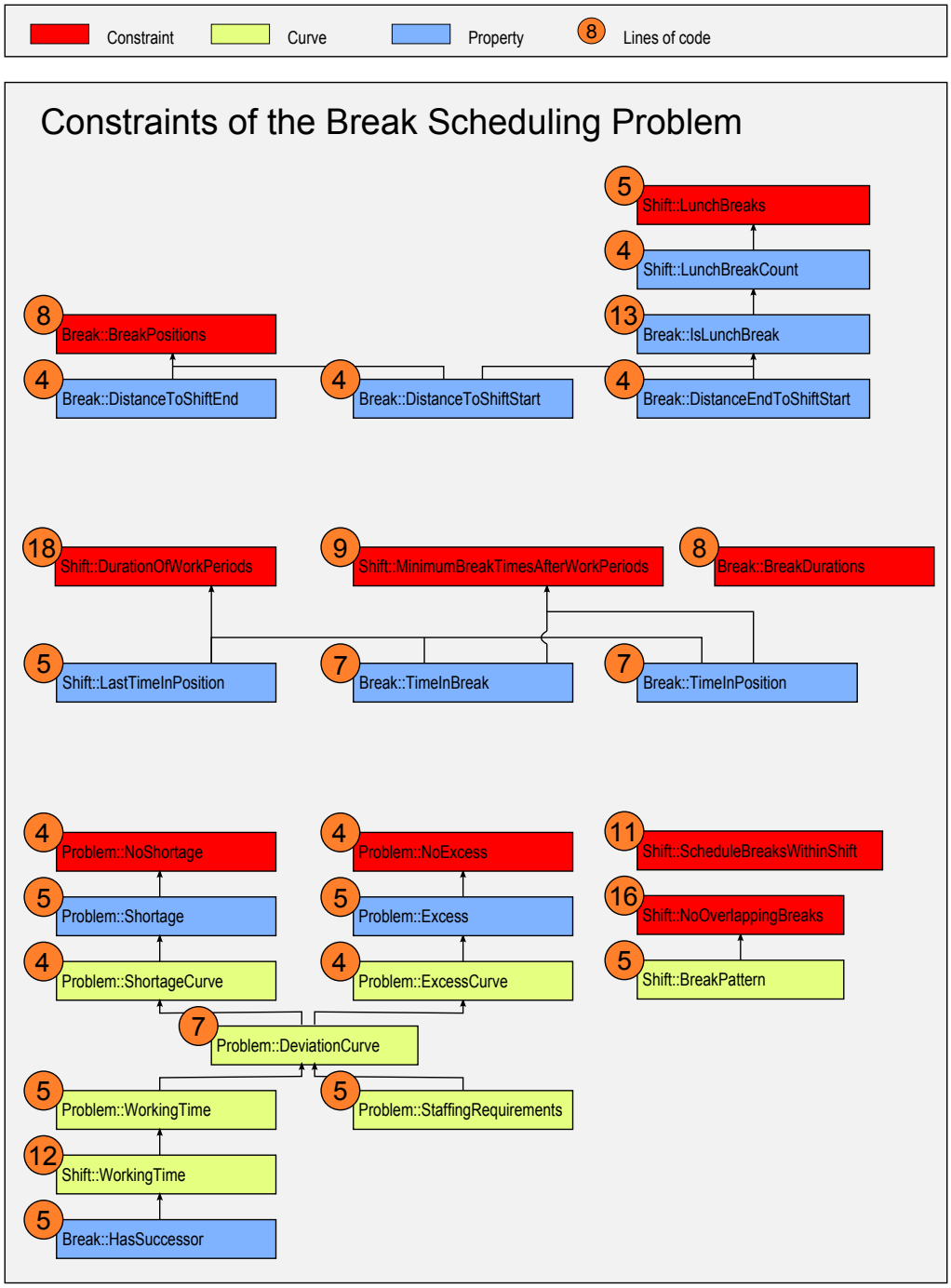


Figure 8.1: TEMPLE model of the constraints involved in the break scheduling problem for supervisory personnel.

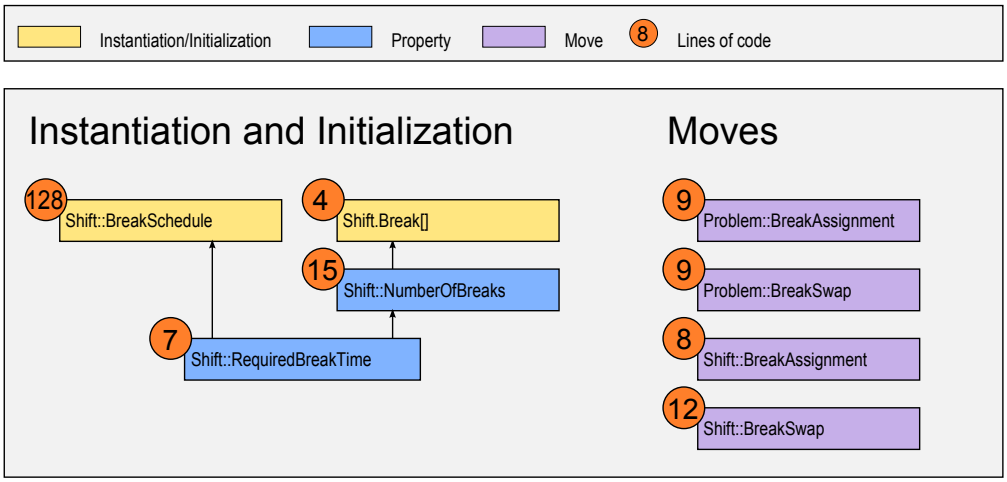


Figure 8.2: TEMPLE model for instantiation elements, initialization elements, and moves involved in the break scheduling problem for supervisory personnel.

Initialization and Instantiation

The initial solution for our problem is obtained with the heuristic proposed in Chapter 3 involving simple temporal problems (STPs) [16]. The applied heuristic constructs an initial break pattern satisfying all hard constraints, i.e., C_1 *Break Positions* - C_5 *Break Durations*, and constraint *NoOverlappingBreaks* and *ScheduleBreaksWithinShift*. All TEMPLE elements defined to instantiate and initialize intervals are shown in the appendix of this thesis, A.11.

Moves

In our TEMPLE program we implemented two moves, the first assigns a new start to a single break, the second one swaps two breaks of different duration within the same shift. We further implemented two variations of these moves, which apply these changes only to breaks placed within understaffed regions. The four moves definitions are presented within the appendix of this thesis, A.12.

8.1.1 Conclusions Drawn from the TEMPLE Model

From our TEMPLE model for the break scheduling problem for supervisory personnel we can draw the following conclusions:

- ▷ In total we needed one man-week to develop a suitable TEMPLE model for the break scheduling problem for supervisory personnel. The resulting TEMPLE program consists only of 500 lines of code and it is formulated in a very modular style. In contrast around 6000 lines of code were needed to implement the min-conflicts based algorithm proposed in Chapter 3. The break scheduling problem for supervisory personnel is modeled in a very compact manner in TEMPLE.
- ▷ Concerning the size of each single TEMPLE element reported in Figure 8.1 and Figure 8.2 we observe that nearly all components could be modeled with fewer than twenty lines of code. On average we needed eleven lines of codes to formulate a TEMPLE element within the TEMPLE program of the break scheduling problem for supervisory personnel. In TEMPLEelements can be formulated very concisely, and the entire problem is built up in a very modular style, as requested by our design goal modularity.
- ▷ Only the formulation of the initialization element `Shift::BreakSchedule` required significantly more effort, namely 128 lines of code. The reason why `Shift::BreakSchedule` is significantly larger in size than any other element is that during within an initialization element we perform actually three tasks: We assign initial values to basic interval properties, we restrict the allowed domain values for basic properties and we link intervals with each other. Therefore, we plan to modify the TEMPLE language, so that each of these three tasks is modeled within a single element. In that way we hope to further increase the modularity of TEMPLE and the simplicity of single language elements.

8.1.2 Computational Results

From the TEMPLE model for the break scheduling for supervisory personnel we created an iterated local search algorithm with our TEMPLE compiler. To evaluate the obtained algorithm we applied it to the twenty real-life benchmark instances, and 10 randomly generated instances for the break scheduling problem for supervisory personnel. For each instance we performed ten runs of the iterated local search algorithm. Each run was carried out under the same conditions as the min-conflicts based algorithm from Chapter 3, namely on a Genuine Intel T2400 laptop running at 1.8 GHz with 2 Gbytes of RAM. A single test run was executed with a one-hour time limit.

Table 8.1 reports the best and mean objective function value and the corresponding standard deviation of the initial solutions and the final results obtained by the iterated local search algorithm in ten runs. Moreover, column MCRW of Table 8.1 presents again the results obtained with the min-conflicts based algorithm. We see that the iterated local search algorithm is able to improve the quality of the initial solutions significantly during the optimization. However, if we compare the results achieved by the iterated local search algorithm with those of the min-conflicts based algorithm we see that the min-conflicts based algorithm outperforms the iterated local search algorithm in every instance.

The min-conflicts based algorithm performs better than the iterated local search algorithm because it was customized toward the break scheduling problem for supervisory personnel. With TEMPLE we introduce an additional programming language layer and that layer causes a certain computational overhead that can be avoided at a lower level implementation of an algorithm. However, the advantage of TEMPLE is that we are able to model problems faster and in that way we can reduce the effort for developing local search algorithms significantly.

Figure 8.3 presents parts of the best solutions obtained for the three benchmark instances, 2fc04a, 3si2ji2 and 50fc04a. Staffing requirements are shown as a red curve, working employees are depicted as a blue curve over time. From the curves for solution 2fc04a and 3si2ji2 we see that shortage of employees could be avoided completely whereas in the presented part of the solution for instance 50fc04a shortage occurs only in three time slots. We conclude that with our TEMPLE model of the break scheduling problem for supervisory personnel we are able to compute solutions of acceptable quality in reasonable time.

	STP-Initial			TEMPLE			MCRW		
Instance	Best	Mean	SD	Best	Mean	SD	Best	Mean	SD
2fc04a	12772	13087.6	283.1	3550	3671.2	66.6	3112	3224.2	86.1
2fc04a03	13178	13529.6	334.0	3506	3676.4	93.4	3138	3199.6	38.7
2fc04a04	12938	13313.6	325.6	3658	3790.8	135.1	3234	3342.1	59.5
2fc04b	12996	13862.4	383.7	2574	2745.6	120.6	1822	2042.8	99.1
3fc04a	12938	13335.2	295.5	2404	2553.0	204.5	1644	1767.0	101.6
3fc04a03	13314	13762.8	314.0	2292	2461.6	81.9	1670	1759.2	53.1
3fc04a04	12680	13368.8	345.1	2464	2586.4	83.1	1932	1980.2	40.4
3si2ji2	10584	10986.0	310.2	3688	3741.4	29.0	3646	3666.6	14.5
4fc04a	12518	13388.0	484.2	2266	2437.8	97.9	1730	1817.1	48.2
4fc04a03	13176	13696.8	400.6	2264	2400.6	126.1	1748	1834.2	55.5
4fc04a04	13086	13687.2	359.4	2404	2542.6	66.7	1982	2063.6	62.3
4fc04b	12000	12888.0	422.4	1950	2089.4	86.8	1410	1489.2	48.7
50fc04a	13518	13999.2	279.2	2580	2740.0	87.1	1672	1827.3	80.6
50fc04a03	13572	14059.2	407.7	2386	2681.2	136.9	1686	1813.2	84.1
50fc04a04	13462	13967.2	373.4	2606	2772.8	106.1	1792	1917.2	64.1
50fc04b	14030	14897.6	449.0	2952	3053.4	92.8	1822	1953.9	77.1
51fc04a	14172	14517.6	223.6	2932	3169.0	167.1	2054	2166.2	62.3
51fc04a03	14176	14515.6	226.8	2898	3017.4	81.4	1950	2050.4	86.5
51fc04a04	14072	14747.6	460.6	2990	3211.0	136.7	2116	2191.4	53.1
51fc04b	14366	15077.6	311.3	3212	3484.6	148.8	2244	2389.4	93.9
Random1-1	15060	15464.4	291.9	1144	1281.2	108.8	728	972.4	176.9
Random1-13	14112	14462.4	247.1	2222	2473.2	177.8	1654	1994.0	172.1
Random1-2	16716	17025.6	218.1	3568	3973.2	235.8	1284	1477.0	99
Random1-24	15600	15814.8	171.3	1804	2129.0	260.0	860	1077.2	153.9
Random1-28	14400	14575.2	109.6	1984	2313.6	224.1	1358	1658.0	212.8
Random1-5	14652	16090.8	571.3	1778	2097.8	228.7	1264	1535.2	245.2
Random1-7	15072	15232.8	132.3	2310	2629.0	163.5	1586	1712.8	74.5
Random1-9	14292	14778.0	282.9	2474	2760.2	216.4	1710	2020.0	233
Random2-1	18540	18990.0	363.1	4146	5129.8	396.7	1686	1855.2	142.1
Random2-4	17412	17786.4	267.2	3864	4801.6	402.5	1712	2052.8	242

Table 8.1: Test results for the iterated local search algorithm generated with our TEMPLE compiler and for the min-conficts-random-walk algorithm from Section 3.

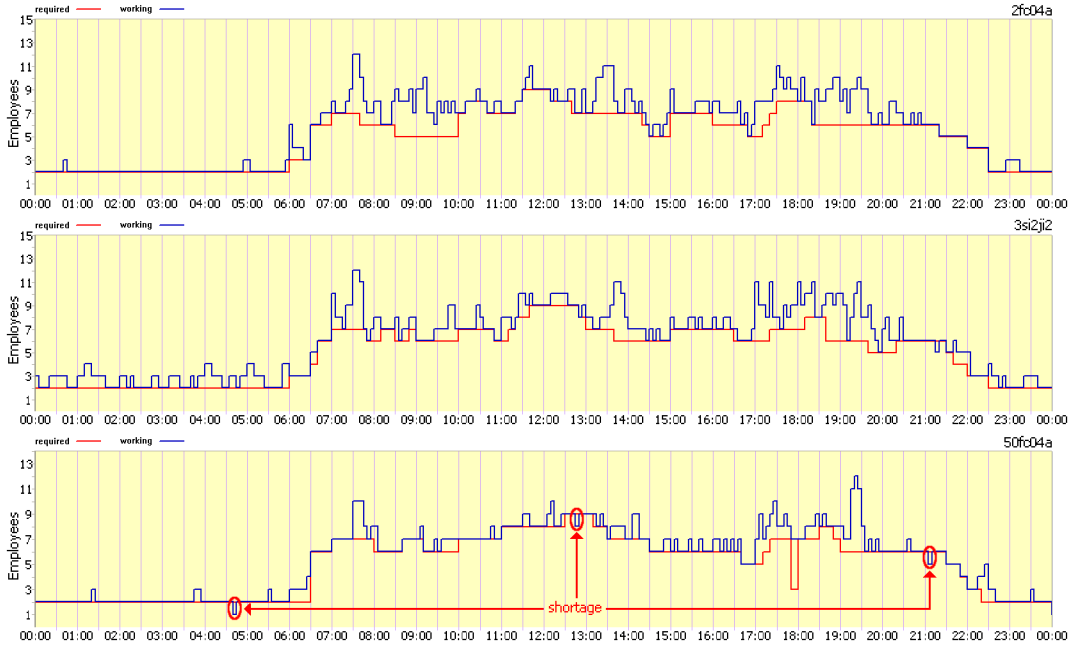


Figure 8.3: Staffing requirements and curve of working employees for parts of the best solutions obtained for the real-life benchmark instances 2fc04a, 3si2ji2 and 50fc04a.

8.2 A Real-life Break Scheduling and Task Assignment Problem

TEMPLE has already been applied successfully in a commercial staff scheduling tool. We built this tool in a research project together with the consulting company Ximes Corp. for one of their customers. In the considered staff scheduling problem we are given task requirements for an entire day, an already existing shift plan and the qualifications of each employee. To obtain a solution we must again compute a break schedule which is completely consistent with a set of legal requirements. In addition, we must also assign the required tasks to available employees in accordance with their qualifications.

8.2.1 Problem Definition

Formally the problem has the following inputs:

- ▷ A *planning period* which is formed by T consecutive time slots $[a_1, a_2), [a_2, a_3), ..., [a_T, a_{T+1}]$, all having the same length, typically 10 minutes. Time points a_1 and $a_T + 1$ represent the beginning and the end of the planning period.
- ▷ A *shift plan* consisting of n shifts $(s_1, s_2, ..., s_n)$. Each shift represents a single employee working within the planning period.
- ▷ The *task requirements*, i.e., the tasks to be performed during the planning period, which are defined as follows. For each time slots $[a_t, a_{t+1})$ we are given a set of tasks $Task_t$ that must be performed during that time slot. A single task must be carried out by a group of two employees. One employee is the head of the group whereas the other acts as the head's assistant.
- ▷ The *qualifications* of each employee $(Q_1, Q_2, ..., Q_n)$. The qualifications of the i-th employee Q_i is a list, specifying which tasks the i-th employee is allowed to carry out as head or as assistant. For instance, if Q_i contains the entry $TASK1\ H$, the i-th employee will be qualified to perform task $TASK1$ as head of the group. The entry $TASK1\ A$ indicates that the i-th will be qualified to carry out task $TASK1$ as an assistant.

To obtain a solution for the considered staff scheduling problem we actually have to achieve the following two goals:

1. We must compute a break schedule for all employees. The obtained break schedule must meet the criteria resulting from an agreement between the workers council and the management board. These constraints must be satisfied completely in order that a break schedule can be applied in practice.

2. We must assign tasks to employees. For each time slot and for each required task we must determine two employees having the correct qualifications as a head and as an assistant. Obviously, we may only assign employees not having a break at the considered time slot. In each time slot we want to perform as many tasks as possible. However, the resulting task assignment should also satisfy several constraints, reflecting ergonomic criteria.

Figure 8.4 shows an artificial sample problem instance for the considered problem. As input we are given a planning period from `08:00` to `18:00`, a shift plan representing six employees, each employee's qualification list and the tasks requirements for the entire planning period. Moreover, Figure 8.4 also presents a solution for the depicted problem instance. The breaks scheduled in each shift are consistent with the agreement between the works committee and the management board. During their working time each employee is assigned tasks in accordance with his qualifications. At any point of time, each required task is performed by two employees one acting as head and the other as assistant. From Figure 8.4 we also see that between `13:30` and `15:00` two employees have additional rest periods. This is due to the fact that in that period four employees are available but only one task is required to be carried out. In the following we want to present the constraints imposed on the break pattern and task assignment in our real-life staff scheduling problem.

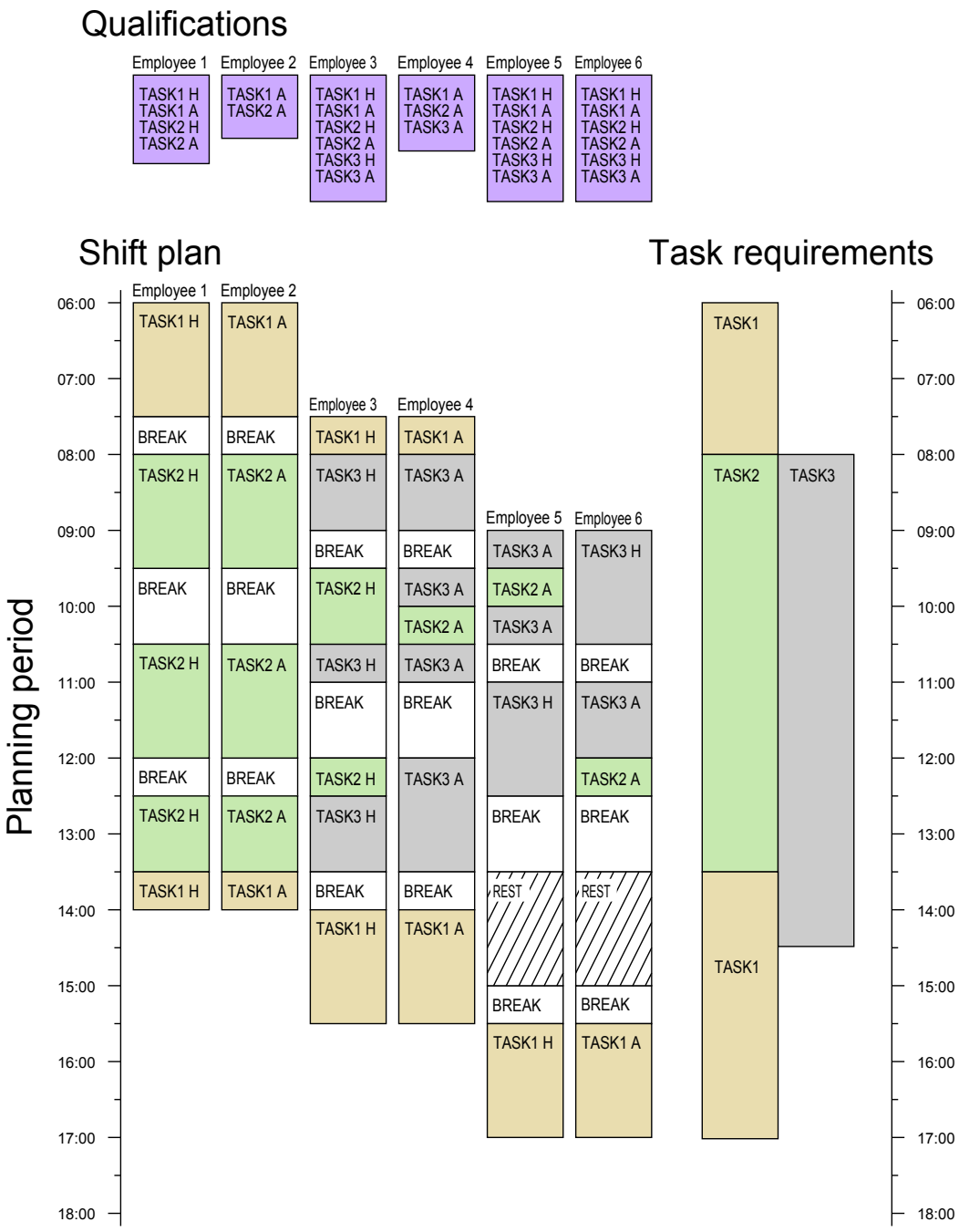


Figure 8.4: An artificial sample instance of the break scheduling and task assignment problem.

Constraints on the Break Schedule

No Overlapping Breaks: Two breaks scheduled within the same shift must not overlap with each other.

Schedule Breaks Within Their Shifts: Each break must lie entirely within the shift it is scheduled.

Minimum Break Time: Each shift must contain at least a minimum percentage of break time, *minimum break time*.

Break Durations: Each break must last at least a certain number minutes, *minimum break duration*.

Lunch Breaks: Each shift must contain at least one lunch break of a certain length, *minimum lunch break duration*.

Legal Break Pattern: Generally, each employee may work longer than *maximum working time* without having a break. Once per shift the duration of a work period can be extended up to *exceptional working time* minutes. This exceptional work period must be followed by a lunch break.

Schedule Blocked Break in Night Shift: A shift starting before and ending after midnight is considered to be a night shift. Each night shift must contain a so-called blocked break which lasts *blocked break duration* minutes.

Constraints on the Task Assignment

Perform Required Tasks: In each time slot $[a_t, a_{t+1})$ of the planning period all required tasks $Task_t$ should be carried out.

Minimum Task Time: A single employee should continuously perform the same task for at least *minimum task time* minutes.

Avoid Task Changes: Employees should not change their task without having a break or being on an additional rest period.

Avoid Rest Periods at Shift Borders: An employee should not have a rest period at the start or the end of its shift.

Training Units

To extend the qualifications of employees, training units can be entered into a shift plan. During a training unit a trainee is instructed by a trainer how to handle a task for which the trainee is not qualified yet. Trainer and trainee carry out together one single task, either as head or as assistant. The last time slot of a training unit is used to review the training. Afterwards both trainer and trainee are assigned a break. Figure 8.5 shows an exemplary training unit including a review followed by a break. To plan training units correctly our problem is extended by the following constraints:

Avoid Breaks in Trainings: Breaks must not be scheduled during training units.

Schedule Break After Training: Each training unit must be followed by a break.

Train Correct Tasks: Trainer and trainee must be assigned the task that is trained during the training unit. Each training unit ends with ten minutes review time.

Intra-Day Absences

A further detail that we had to consider in our problem were intra-day absences of employees. The reasons for intra-day absences of employees are manifold: employees may participate in meetings, they may have a doctor's appointment, employees may fall ill, etc. In a correct solution for our problem breaks must not be scheduled during intra-day absences:

Avoid Breaks in Absences Breaks must not be scheduled during intra-day absences of employees.

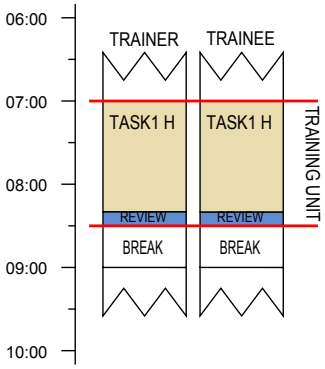


Figure 8.5: Training and subsequent review for task $TASK1\ H$ followed by a break.

8.2.2 A Three-Phase Approach

Due to different kinds of requirements and constraints the considered problem is very complex as a whole. For this reason we decided to decompose the entire problem into three separate phases each of which is modeled and solved by a separate TEMPLE program:

1. **Break Schedule Initialization.** We compute a legal break schedule which is consistent with all constraints imposed on a break pattern, training units and intra-day absences.

2. **Break Schedule Optimization.** From the given task requirements we derive simpler staffing requirements. For each task to be performed during a specific time slot we require that two people must be working at that time. Then we optimize the break schedule according to these staffing requirements, whereby we ensure that the break schedule remains always legal during and after this optimization phase. The qualifications of employees are not considered in this step.

3. **Task Assignment and Optimization.** For each time slot we assign the required tasks to available employees heuristically. Afterwards we further try to reduce the violations of constraints imposed on the task assignment to a minimum degree. The break schedule is not changed further during this phase, thus break schedule remains still legal. In the obtained task assignment employees carry out as many tasks as possible. If any, only a few soft constraints concerning additional rest periods, tasks changes or minimum task times, are violated after the last optimization phase.

Table 8.2 presents an overview on the hard and soft constraints involved in each phase's TEMPLE program.

8.2.3 Phase I - Break Schedule Initialization

In phase I we want to obtain a legal break schedule satisfying all constraints on training units and absences. To obtain an initial solution in phase I for each shift we determine the minimum amount of break time to be scheduled and further derive the number of breaks to be instantiated. In each shift the break time is distributed among lunch breaks, ordinary breaks and each night shift gets a blocked break. In each day shift we obtain an initial break pattern by solving the simple temporal problem (STP) resulting from constraint *Legal Break Pattern*:

Constraint	Phase I	Phase II	Phase III
Schedule Breaks Within Their Shifts	hard	hard	hard
Minimum Break Time	hard	hard	hard
Lunch Breaks	hard	hard	hard
Break Durations	hard	hard	hard
Schedule Blocked Break in Night Shift	hard	hard	hard
No Overlapping Breaks	hard	hard	hard
Legal Break Pattern	soft	hard	hard
Avoid Breaks In Trainings	soft	hard	hard
Schedule Break After Training	soft	hard	hard
Avoid Breaks In Absences	soft	hard	hard
Reduce Shortage	-	soft	-
Train Correct Tasks	-	-	hard
Minimum Task Time	-	-	soft
Avoid Task Changes	-	-	soft
Avoid Rest Periods at Shift Borders	-	-	soft

Table 8.2: Overview on the constraints involved in the three phases.

$$\begin{aligned} b_1.Start - s_i.Start &\in [\textit{minimum task time}, \textit{maximum working time}] \\ b_{j+1}.Start - b_j.End &\in [\textit{minimum task time}, \textit{maximum working time}] \\ s_i.End - b_m.End &\in [\textit{minimum task time}, \textit{maximum working time}] \end{aligned}$$

where $(b_1, ..., b_j, b_{j+1}, ..., b_m)$ are the breaks of shift s_i in temporal order. For night shifts we solve a similar STP that also takes blocked four hour breaks into account. By solving the simple temporal problems for each shift we obtain a break pattern which is consistent with all hard constraints of phase I (see Table 8.2). Since breaks may be scheduled in absence times or training units, or there may be trainings which are not followed by a break, the solutions generated via STPs may violate the soft constraints *Avoid Breaks In Trainings*, *Schedule Break After Training* and *Avoid Breaks In Absences* .

To eliminate these constraint violations we implemented several moves, depicted in Figure 8.6, changing a single shift's break pattern in our TEMPLE program for phase I. The move in Figure 8.6 (a) repositions a single break in its associated shift, the move shown in Figure 8.6 (b) shifts the entire break pattern for a single employee, and the move in Figure 8.6 (c) swaps two breaks of different duration.

However, to eliminate constraint violations in shifts with training units effectively, we introduced additional moves. Figure 8.6 (d) shows a move which detects a break scheduled near a training unit and places them right after the training. The move presented in Figure 8.6 (e) recognizes a break scheduled illegally in a training unit, and resolves that constraint violation by moving the break outside the training. Finally, the move illustrated in Figure 8.6 (f) tries to compute a new break pattern by solving a simple temporal problem, which also considers training units.

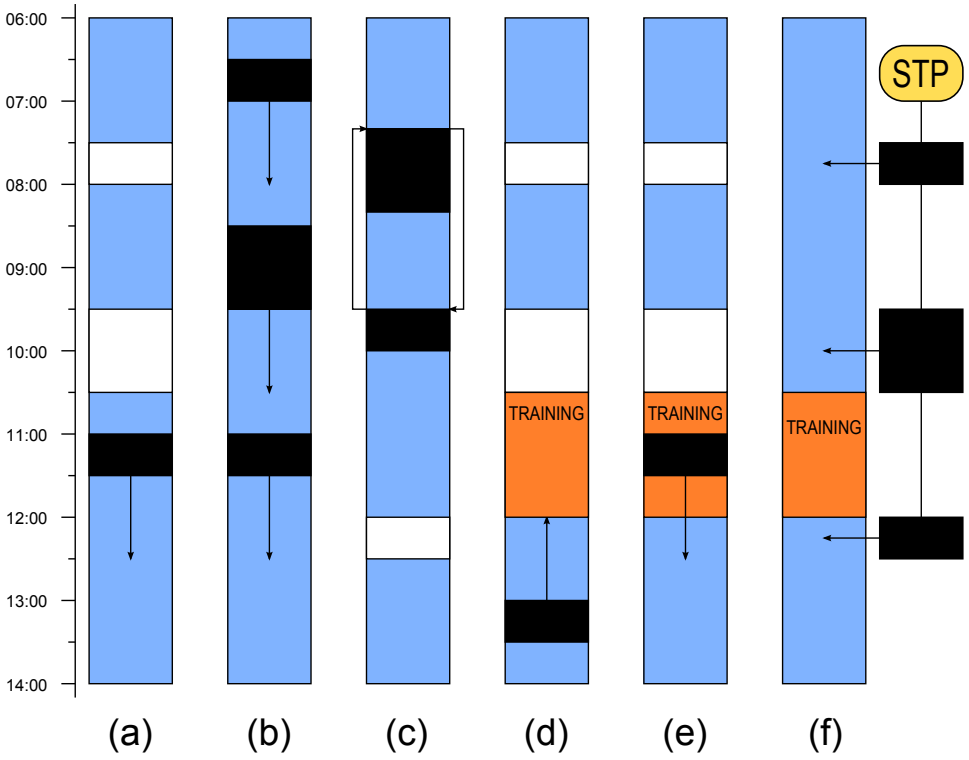


Figure 8.6: Moves applied in phase I to obtain a legal break pattern.

8.2.4 Phase II - Break Schedule Optimization

In phase II we want to optimize the break schedule in order that at any time enough people are available to carry out the required tasks. For that purpose we extend the TEMPLE program of phase I in the following manner:

1. From the task requirements, we derive a curve representing staffing requirements. For each required task this curve is incremented by two units because one task actually must be performed by two employees, one head and one assistant.

2. From the staffing requirements, shifts, and breaks, we can further derive the deviation of staffing requirements for a specific solution, the shortage of employees caused by a specific break schedule, and finally we impose the soft constraint *Reduce Shortage*. Reducing the shortage of employees is the single objective in phase II.

3. All constraints on the break schedule, training units and absence times from phase I are reused again in phase II. However, all constraints that were soft constraints in phase I are changed into hard constraints in phase II, as can be seen in Table 8.2. In that way we ensure that the break schedule obtained in phase I still remains legal in phase II.

The break schedule computed in phase I acts as the initial solution of phase II. To reduce shortage of employees we re-employ the three moves presented in Figure 8.6 (a)-(c) which we have already used in phase I. In addition, we implemented two additional moves, the first one, shown in Figure 8.7 (a) swaps the position of two breaks scheduled in two distinct shifts. The second additional move, presented in Figure 8.7 (b), tries to eliminate shortage of employees by changing break positions across shifts. This move selects a time slot with shortage of employees and identifies a break starting or ending in that time slot. This break is shifted by one time slot, and with that break also the shortage is shifted to a new time slot in the planning period. The moves tries to shift breaks and corresponding shortages until a break is moved to a task with excess of employees. In that manner shortage and excess are merged and eliminated.

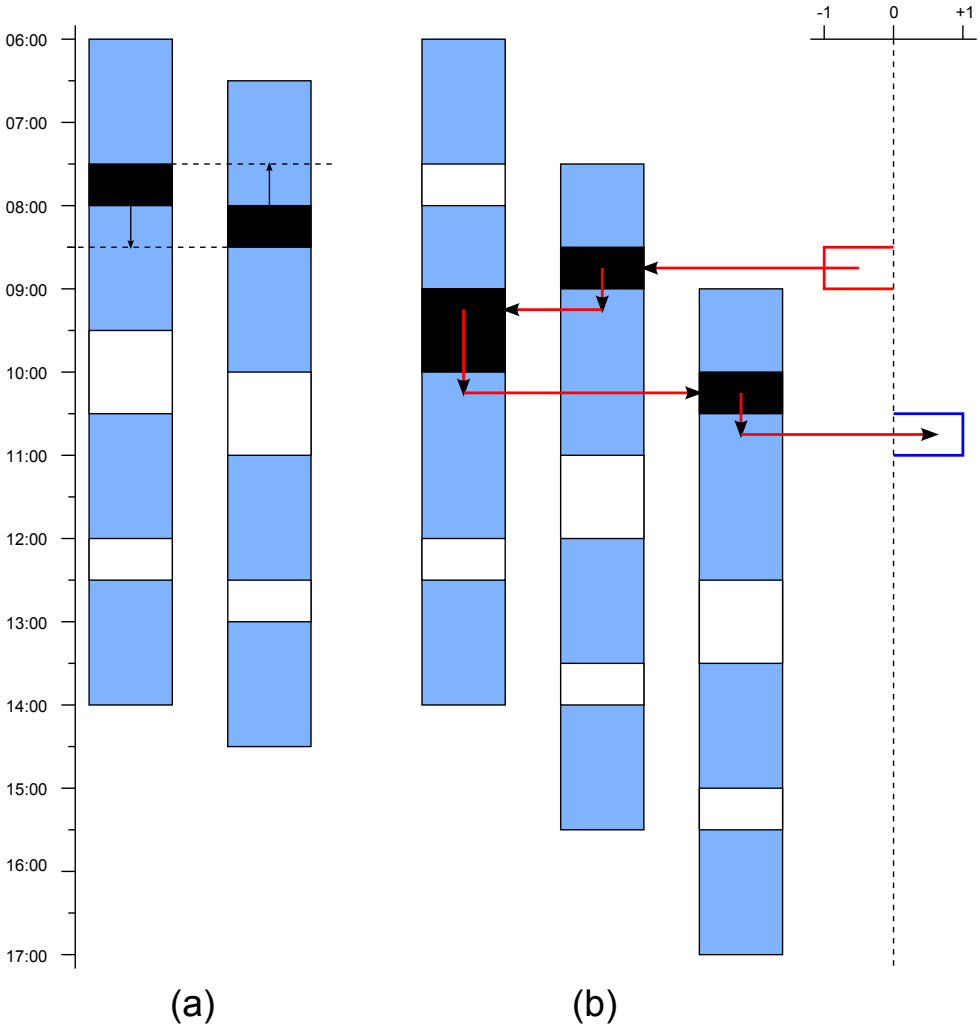


Figure 8.7: Additional moves applied in phase II to optimize a break schedule.

8.2.5 Phase III - Task Assignment and Optimization

Modeling the Task Assignment Problem as Integer Program

In each time slot we want to assign as many tasks as possible to working employees. Figure 8.8 shows how this problem can be modeled as integer program:

- ▷ Let $Task_t = \{task_1, task_2, ..., task_m\}$ be the set of tasks required at time slot $[a_t, a_{t+1})$.
- ▷ For each task in $Task_t = \{task_1, task_2, ..., task_m\}$ we introduce a binary variable $task_j$. In a solution $task_j = 1$ if and only if the task corresponding to $task_j$ is carried out.
- ▷ For each working employee i having the qualification to perform a required task $task_j$ acting as a head of group we introduce a binary variable e_{ij}^H. In a solution $e_{ij}^H = 1$ if and only if employee i carries out $task_j$ as a head of group. We define E_i^H to be a set containing all variables e_{ij}^H which have been introduced for employee i and T_j^H to be the set of all variables e_{ij}^H that have been introduced for a single task $task_j$.
- ▷ For each working employee i qualified to perform a required tasks $task_j$ as a head's assistant we introduce a binary variable e_{ij}^A. In a solution $e_{ij}^A = 1$ if and only if employee i performs the task corresponding to $task_j$ as an assistant. We define E_i^A to be the set containing all variables e_{ij}^A which have been introduced for employee i and T_j^A to be the set of all variables that have been introduced for a single task $task_j$.
- ▷ We introduce several restrictions to avoid that one task is carried out by several heads or assistants (1)-(2), or that a single employee is assigned several tasks (3). Moreover we state that each task must be carried out by a group of two employees, one acting as the head of group, the other as the head's assistant (4).
- ▷ As an objective we want to maximize the number of assigned tasks.

Initial Solution

By solving the IP-problems arising in each time slot we can guarantee that as many tasks as possible can be assigned. However, already within an initial task assignment, we want prevent employees from constantly changing their assigned tasks. For that

$$
\begin{array}{lll}
\max & \sum_{j=1}^{m} task_j & \\
\text{s. t.} & & \\
(1) & \sum_{e_{ij}^H \in T_j^H} e_{ij}^H & \leq 1 \qquad j = 1, ..., m \\
(2) & \sum_{e_{ij}^A \in T_j^A} e_{ij}^A & \leq 1 \qquad j = 1, ..., m \\
(3) & \sum_{e_{ij}^H \in E_i^H} e_{ij}^H + \sum_{e_{ij}^A \in E_i^A} e_{ij}^A & \leq 1 \qquad i = 1, ..., n \\
(4) & \sum_{e_{ij}^H \in T_j^H} e_{ij}^H + \sum_{e_{ij}^A \in T_j^A} e_{ij}^A & = 2task_j \qquad j = 1, ..., m \\
(5) & task_j, e_{ij}^H, e_{ij}^A \in \{0, 1\} &
\end{array}
$$

Figure 8.8: To obtain an initial task assignment, we solve an integer problem in each time slot to guarantee that as much tasks as possible are carried out.

purpose we obtain the initial solution of phase III with a heuristic approach trying to reassign employees the same tasks again and again.

For each time slot $[a_t, a_{t+1})$ we assign tasks to employees as follows:

1. We determine the maximum number of tasks $Task_{max}$ that can be assigned.
2. We try to solve a more restricted integer program: Each employee must be assigned the same task again which he or she has carried out in the previous time slot. Employees who had a break or rest period at the previous time slot can be assigned any task for which they are qualified. If it's still possible to carry out $Task_{max}$ tasks we will apply the obtained task assignment.
3. Otherwise we iteratively select an employee and allow him to carry out any tasks he is qualified for until we can carry out $Task_{max}$ tasks. An employee changing his task is selected as follows:
   (a) If possible we select an employee that has carried out the same task for at least *minimum task time* minutes and has at least *minimum task time* minutes left until the next break starts or the employee's shift ends.
   (b) Otherwise we select an arbitrary employee.

We implemented the proposed heurisitic as an `Initialize` element in the TEMPLE model for phase III, and used an IP-solver integrated in Comet, to solve integer programs arising during initialization.

Constraints and Moves

To improve the initial solution returned by our heuristic we modified the TEMPLE program of phase II for our needs. Since the break schedule is not changed while optimizing the task assignment, we removed the soft constraint *Reduce Shortage*. Moreover, we added a hard constraint *Train Correct Tasks* to check that at any time employees are assigned only those tasks which are actually required. Furthermore, we imposed three additional soft constraints *Minimum Task Time*, *Avoid Task Changes*, and *Avoid Rest Periods at Shift Borders*, to optimize the task assignment even further. Table 8.2 presents all hard and soft constraints defined for phase III.

Figure 8.9 sketches the moves we developed to prevent employees from performing a single task for less than *Minimum Task Time* minutes. These moves analyze task assignments, detect critical points where tasks are assigned for less than *minimum task time* minutes to an employee, and exchange assigned tasks between several employees. In addition, we implemented moves to reduce task changes and to avoid rest periods assigned at shift borders. These moves are presented in Figure 8.10.

Figure 8.9: Moves eliminating situations in which employees carry out a single task for less than *minimum task time* minutes.

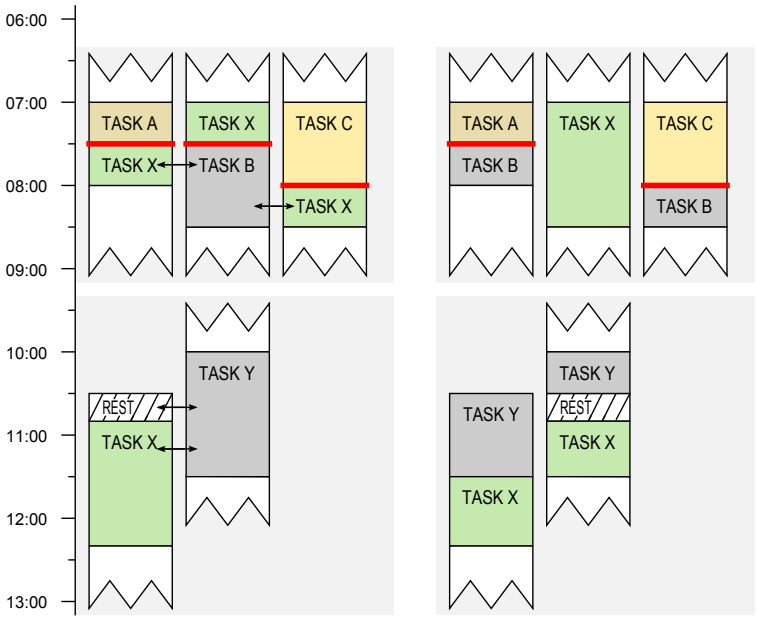


Figure 8.10: Moves reducing task changes and rest periods at shift borders.

8.2.6 Break Scheduling and Task Assignment Tool

The three phase optimization algorithm described in the previous section is included in a commercial break scheduling and task assignment tool. We built this tool in a project together with the consulting company Ximes Corp. for one of their customers. The goal of the project was to develop a working prototype implementing our proposed three phase approach. With that prototype we wanted to deliver a proof of concept that automated break scheduling and task assignment is possible in TEMPLE in reasonable time.

Basically, the working prototype consists of two components: The first component is a Microsoft Excel VBA application which gathers and processes user input and visualizes the computed results. The second part is the three phase optimization algorithm generated with TEMPLE computing a feasible break pattern and a high-quality task assignment. Currently, the prototype is used by human decision makers on customer's site to manage and handle the following three cases:

1. The prototype is used to calculate a break schedule and task assignment at the start of a working day. At this stage the decision maker might place training units to be held in overstaffed periods and he or she places intra-day absences which are already known in advance.

2. The prototype is used to asses the consequences of intra-day changes. Whenever the task requirements are altered or an employee becomes spontaneously absent, the decision maker triggers the computation of an updated break schedule or task assignment. Breaks that have already been consumed or tasks that have already been performed are not changed by that re-computation but breaks and tasks located in the future are reconfigured to perform as many tasks as possible.

3. The prototype is used to react on intra-day changes. If after a change one or several tasks cannot be performed any longer the decision maker must cancel training units, or deploy additional employees. As a last resort the decision maker can adapt the task requirements.

To illustrate the use cases just mentioned we will present a sample application of our break scheduling and task assignment tool on the sample problem instance presented in Figure 8.4. In that sample problem we are given a shift plan consisting of six employees on duty, each employee's qualifications and the task requirements for a planning period from `06:00` to `17:00`.

Calculating a schedule at the start of a working day

At the start of the working day, the decision maker enters the shift plan and task requirements into our break scheduling and task assignment tool. Since the employees' qualifications do not change very frequently the qualifications are stored in a separated file, where they are accessed by our break scheduling and task assignment tool. Then the decision maker starts the computation of a break schedule and task assignment for the current working day. The screen shot in Figure 8.11 shows how the obtained solution is represented within our break and task assignment tool.

In each 10-minute time slot a single employee is assigned the task which must be performed during that time slot or an employee can be assigned a break (orange rectangle) or rest periods (shaded orange rectangle). On the right hand side there are three columns reporting how many tasks are required, how many are actually assigned to employees, and how many tasks cannot be processed in each time slot. Unprocessed tasks are highlighted with red color. In Figure 8.11 we see the task requirements are satisfied completely.

Above the break and task schedule our tool reports the break time and rest period percentages assigned in each employee's shift. From these percentages a decision maker can check whether enough break time is assigned to each single employee. From high rest period percentages the decision maker can conclude that the current solution is overstaffed, i.e., in certain periods actually more people than required are working.

In the schedule presented in Figure 8.11 the task requirements are satisfied completely and the assigned break times and the break schedule comply with all legal requirements. Consequently, the obtained solution could be deployed in practice.

Considering the schedule presented in Figure 8.11 the decision maker recognizes that in the period from `12:30` until `15:30` the obtained solution is overstaffed. In that time range rest periods are assigned very frequently to employees. Therefore, the decision maker exploits that excess of employees and inserts a training unit starting at `12:00` and ending at `13:30` into the schedule. In that training unit employee `E3` should train employee `E4` on task `TASK3 H`. After entering the training unit he computes a new solution from the scratch, part of which is presented in Figure 8.12. In the obtained solution both trainer and trainee are assigned `TASK3 H` in the period from `12:00` until `13:20` whereas the last ten minutes of the training unit are used to review the training. On the right hand side we see that the task requirements are still satisfied. Therefore, the obtained schedule will be applied during the current working day.

Break Scheduler and Task Assigner [Calculate schedule] [Update schedule]

Break Time	##.#%	##.#%	##.#%	##.#%	##.#%	##.#%			Tasks		
Rest Time	0.0%	6.3%	0.0%	0.0%	12.5%	18.8%					
Employees	E1	E2	E3	E4	E5	E6			Required	Assigned	Unassigned
05:30									0	0	0
05:40									0	0	0
05:50									0	0	0
06:00	TASK1 H	TASK1 A							1	1	0
06:10	TASK1 H	TASK1 A							1	1	0
06:20	TASK1 H	TASK1 A							1	1	0
06:30	TASK1 H	TASK1 A							1	1	0
06:40	TASK1 H	TASK1 A							1	1	0
06:50	TASK1 H	TASK1 A							1	1	0
07:00	TASK1 H	TASK1 A							1	1	0
07:10	TASK1 H	TASK1 A							1	1	0
07:20	TASK1 H	TASK1 A							1	1	0
07:30			TASK1 H	TASK1 A					1	1	0
07:40			TASK1 H	TASK1 A					1	1	0
07:50			TASK1 H	TASK1 A					1	1	0
08:00	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
08:10	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
08:20	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
08:30	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
08:40	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
08:50	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
09:00	TASK2 H			TASK3 A	TASK3 H	TASK2 A			2	2	0
09:10	TASK2 H			TASK3 A	TASK3 H	TASK2 A			2	2	0
09:20	TASK2 H			TASK3 A	TASK3 H	TASK2 A			2	2	0
09:30		TASK2 A	TASK2 H		TASK3 H	TASK3 A			2	2	0
09:40		TASK2 A	TASK2 H		TASK3 H	TASK3 A			2	2	0
09:50		TASK2 A	TASK2 H		TASK3 H	TASK3 A			2	2	0
10:00		TASK2 A	TASK2 H		TASK3 H	TASK3 A			2	2	0
10:10		TASK2 A	TASK2 H		TASK3 H	TASK3 A			2	2	0
10:20		TASK2 A	TASK2 H		TASK3 H	TASK3 A			2	2	0
10:30	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
10:40	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
10:50	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
11:00	TASK2 H	TASK2 A		TASK3 A		TASK3 H			2	2	0
11:10	TASK2 H	TASK2 A		TASK3 A		TASK3 H			2	2	0
11:20	TASK2 H	TASK2 A		TASK3 A		TASK3 H			2	2	0
11:30			TASK2 H	TASK3 A	TASK2 A	TASK3 H			2	2	0
11:40			TASK2 H	TASK3 A	TASK2 A	TASK3 H			2	2	0
11:50			TASK2 H	TASK3 A	TASK2 A	TASK3 H			2	2	0
12:00	TASK2 H		TASK3 A		TASK2 A	TASK3 H			2	2	0
12:10	TASK2 H		TASK3 A		TASK2 A	TASK3 H			2	2	0
12:20	TASK2 H		TASK3 A		TASK2 A	TASK3 H			2	2	0
12:30	TASK2 H		TASK3 H	TASK3 A	TASK2 A				2	2	0
12:40	TASK2 H		TASK3 H	TASK3 A	TASK2 A				2	2	0
12:50	TASK2 H		TASK3 H	TASK3 A	TASK2 A				2	2	0
13:00	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
13:10	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
13:20	TASK2 H	TASK2 A	TASK3 H	TASK3 A					2	2	0
13:30	TASK1 H	TASK1 A							1	1	0
13:40	TASK1 H	TASK1 A							1	1	0
13:50	TASK1 H	TASK1 A							1	1	0
14:00				TASK1 A	TASK1 H				1	1	0
14:10				TASK1 A	TASK1 H				1	1	0
14:20				TASK1 A	TASK1 H				1	1	0
14:30			TASK1 H	TASK1 A					1	1	0
14:40			TASK1 H	TASK1 A					1	1	0
14:50			TASK1 H	TASK1 A					1	1	0
15:00			TASK1 H	TASK1 A					1	1	0
15:10			TASK1 H	TASK1 A					1	1	0
15:20			TASK1 H	TASK1 A					1	1	0
15:30					TASK1 H	TASK1 A			1	1	0
15:40					TASK1 H	TASK1 A			1	1	0
15:50					TASK1 H	TASK1 A			1	1	0
16:00					TASK1 H	TASK1 A			1	1	0
16:10					TASK1 H	TASK1 A			1	1	0
16:20					TASK1 H	TASK1 A			1	1	0
16:30					TASK1 H	TASK1 A			1	1	0
16:40					TASK1 H	TASK1 A			1	1	0
16:50					TASK1 H	TASK1 A			1	1	0
17:00									0	0	0

Figure 8.11: The break and task schedule computed by our break scheduling and task assignment tool for the problem given in Figure 8.4.

Assessing the effect of intra-day changes

At 10:30 the decision maker is told that employee E6 must attend a meeting from 12:30 until 13:30. Since according to the current schedule employee E6 is supposed to be working during that time the decision maker enters the meeting as an intra-day absence and uses our tool to compute an updated schedule for the remaining day starting at 11:00. Figure 8.13 shows the updated schedule returned by the break scheduling and task assignment tool. The meeting has been inserted correctly in the shift for employee E6 between 12:30 and 13:30. However, between 13:20 and 13:30 one task cannot be processed anymore, the corresponding entry in the bar on the right in Figure 8.13 is marked red.

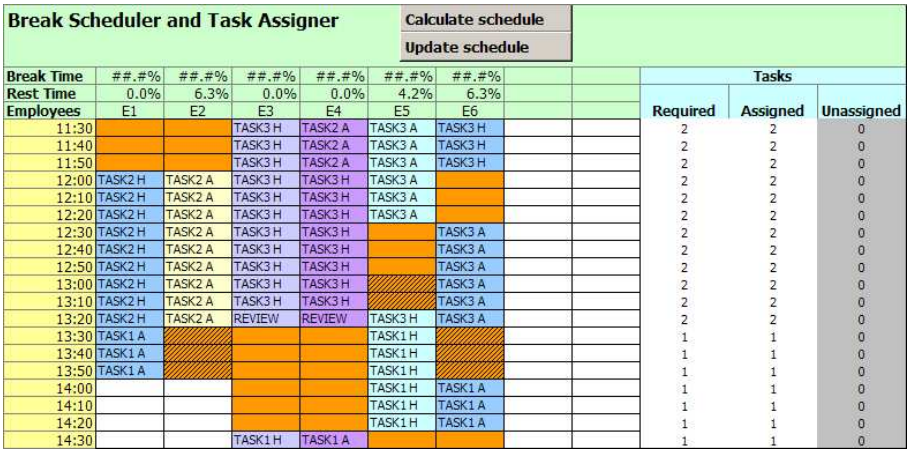
Reacting on intra-day changes

The decision maker must react on the arisen violation of the task requirements. Since the unassigned task coincides with the scheduled training unit, he decides to remove that training unit from the schedule and computes an updated schedule for the remaining day. Figure 8.14 presents the schedule obtained after the removal of the training unit. We see that the violation of task requirements has disappeared completely. Finally, the decision maker informs the employees that their break and task schedule will be changed and hands out the altered schedule to each affected employee.

8.2.7 A Note on the Quality of the Solutions Obtained with the Break Scheduler and Task Assigner

At this point we want to illustrate that the solutions computed by our break scheduling and task assignment tool are of acceptable quality. For that purpose we constructed an artificial problem instance having the comparable features and characteristics as the problems solved at the industrial customer of Ximes Corp. :

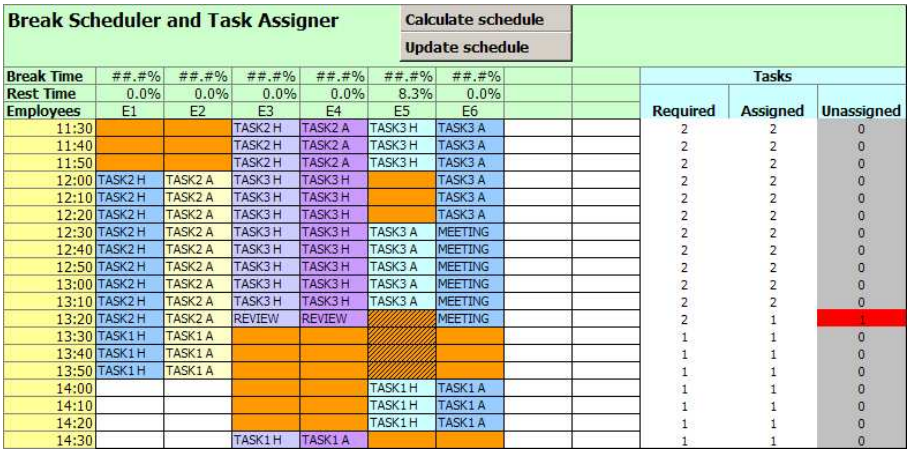
- ▷ The planning period of the considered problem comprises 31 hours, starting at 00:00 and ending at 07:00 of the following day. The time granularity of the planning period is set to 10-minutes time slots.
- ▷ At the beginning of the planning period the task requirements require three tasks to be performed. Then the requirements steadily increase during the morning and between 11:40 and 14:00 they reach their maximum. During this period thirteen tasks must be assigned to available employees. Afterwards the requirements decline again and fall back to three tasks around 23:00.



Break Scheduler and Task Assigner — Calculate schedule / Update schedule

Break Time	##.#%	##.#%	##.#%	##.#%	##.#%	##.#%	Tasks		
Rest Time	0.0%	6.3%	0.0%	0.0%	4.2%	6.3%			
Employees	E1	E2	E3	E4	E5	E6	Required	Assigned	Unassigned
11:30			TASK3 H	TASK2 A	TASK3 A	TASK3 H	2	2	0
11:40			TASK3 H	TASK2 A	TASK3 A	TASK3 H	2	2	0
11:50			TASK3 H	TASK2 A	TASK3 A	TASK3 H	2	2	0
12:00	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A		2	2	0
12:10	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A		2	2	0
12:20	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A		2	2	0
12:30	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
12:40	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
12:50	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
13:00	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
13:10	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
13:20	TASK2 H	TASK2 A	REVIEW	REVIEW	TASK3 H	TASK3 A	2	2	0
13:30	TASK1 A				TASK1 H		1	1	0
13:40	TASK1 A				TASK1 H		1	1	0
13:50	TASK1 A				TASK1 H		1	1	0
14:00					TASK1 H	TASK1 A	1	1	0
14:10					TASK1 H	TASK1 A	1	1	0
14:20					TASK1 H	TASK1 A	1	1	0
14:30			TASK1 H	TASK1 A			1	1	0

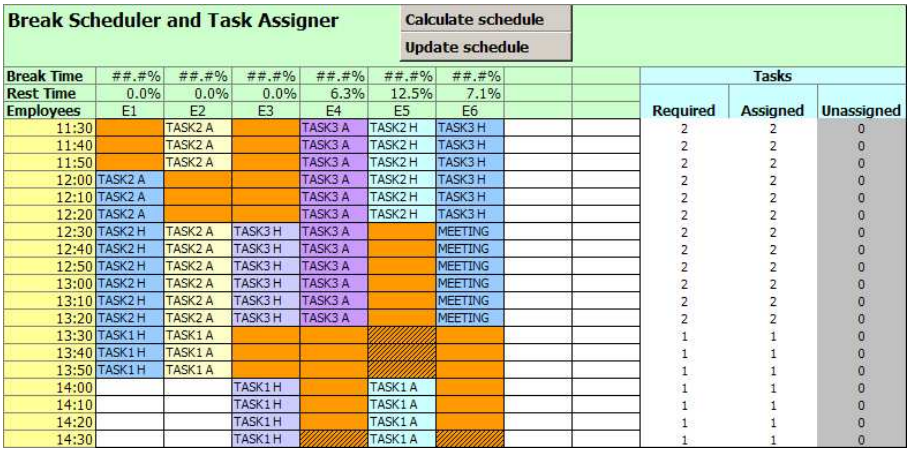
Figure 8.12: Schedule after a training unit has been inserted between 12:00 and 13:30.



Break Scheduler and Task Assigner — Calculate schedule / Update schedule

Break Time	##.#%	##.#%	##.#%	##.#%	##.#%	##.#%	Tasks		
Rest Time	0.0%	0.0%	0.0%	0.0%	8.3%	0.0%			
Employees	E1	E2	E3	E4	E5	E6	Required	Assigned	Unassigned
11:30			TASK2 H	TASK2 A	TASK3 H	TASK3 A	2	2	0
11:40			TASK2 H	TASK2 A	TASK3 H	TASK3 A	2	2	0
11:50			TASK2 H	TASK2 A	TASK3 H	TASK3 A	2	2	0
12:00	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
12:10	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
12:20	TASK2 H	TASK2 A	TASK3 H	TASK3 H		TASK3 A	2	2	0
12:30	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A	MEETING	2	2	0
12:40	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A	MEETING	2	2	0
12:50	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A	MEETING	2	2	0
13:00	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A	MEETING	2	2	0
13:10	TASK2 H	TASK2 A	TASK3 H	TASK3 H	TASK3 A	MEETING	2	2	0
13:20	TASK2 H	TASK2 A	REVIEW	REVIEW		MEETING	2	1	1
13:30	TASK1 H	TASK1 A					1	1	0
13:40	TASK1 H	TASK1 A					1	1	0
13:50	TASK1 H	TASK1 A					1	1	0
14:00					TASK1 H	TASK1 A	1	1	0
14:10					TASK1 H	TASK1 A	1	1	0
14:20					TASK1 H	TASK1 A	1	1	0
14:30			TASK1 H	TASK1 A			1	1	0

Figure 8.13: Schedule after employee E6 must attend a meeting from 12:30 until 13:30.



Break Scheduler and Task Assigner — Calculate schedule / Update schedule

Break Time	##.#%	##.#%	##.#%	##.#%	##.#%	##.#%	Tasks		
Rest Time	0.0%	0.0%	0.0%	6.3%	12.5%	7.1%			
Employees	E1	E2	E3	E4	E5	E6	Required	Assigned	Unassigned
11:30		TASK2 A		TASK3 A	TASK2 H	TASK3 H	2	2	0
11:40		TASK2 A		TASK3 A	TASK2 H	TASK3 H	2	2	0
11:50		TASK2 A		TASK3 A	TASK2 H	TASK3 H	2	2	0
12:00	TASK2 A			TASK3 A	TASK2 H	TASK3 H	2	2	0
12:10	TASK2 A			TASK3 A	TASK2 H	TASK3 H	2	2	0
12:20	TASK2 A			TASK3 A	TASK2 H	TASK3 H	2	2	0
12:30	TASK2 H	TASK2 A	TASK3 H	TASK3 A		MEETING	2	2	0
12:40	TASK2 H	TASK2 A	TASK3 H	TASK3 A		MEETING	2	2	0
12:50	TASK2 H	TASK2 A	TASK3 H	TASK3 A		MEETING	2	2	0
13:00	TASK2 H	TASK2 A	TASK3 H	TASK3 A		MEETING	2	2	0
13:10	TASK2 H	TASK2 A	TASK3 H	TASK3 A		MEETING	2	2	0
13:20	TASK2 H	TASK2 A	TASK3 H	TASK3 A		MEETING	2	2	0
13:30	TASK1 H	TASK1 A					1	1	0
13:40	TASK1 H	TASK1 A					1	1	0
13:50	TASK1 H	TASK1 A					1	1	0
14:00			TASK1 H		TASK1 A		1	1	0
14:10			TASK1 H		TASK1 A		1	1	0
14:20			TASK1 H		TASK1 A		1	1	0
14:30			TASK1 H		TASK1 A		1	1	0

Figure 8.14: Schedule after the removal of a training unit.

- ▷ The input shift plan consists of 63 shifts. The duration of shifts ranges between six and ten hours, the average shift duration is about eight and a half hours.
- ▷ In addition to the shift plan we are given eight night shifts starting at the previous working day. For these night shifts we are already given a break schedule and task assignment that must not be modified by our tool. Since these night shifts coincide with the first seven hours of our input shift plan we must also consider these shifts while generating a break schedule for input shift plan and while assigning tasks to employees.
- ▷ From the 63 deployed employees 53 are allowed to perform any task. The remaining 10 employees may only act as a head's assistant.

We computed a solution for this instance with our break scheduling and task assignment tool on a Genuine Intel T2400 laptop running at 1.8 GHz with 2 Gbytes of RAM. We limited the overall running time to the usual running time at the customer's site, namely five minutes. Since the returned solution is too large to be presented within one figure we divided it into several figures, Figure B.1 - Figure B.3, which are presented in the Appendix of this thesis.

All legal requirements resulting from the agreement between the workers council and the management board are satisfied completely by the obtained solution. Considering soft constraints we observe that the returned solution has the following features:

Unassigned Tasks:	0
Short Task Assignments:	9
Task Changes:	56
Rest Periods at Shift Border:	15

Most important of all, the task requirements are satisfied completely within our solution. Each required task was assigned to a group of employees and can be processed during the working day. Moreover, there are only a few situations in which an employee performs the same task for less than *minimum task time*.

There are 56 immediate task changes in the obtained solution meaning that on average about one immediate task changes per employee occurs. According to our customer, in such large instances tasks changes could not be avoided completely. Moreover, with regard to immediate task changes the solutions computed by our product have the same

quality as the manually constructed ones, which were computed at the costumer's site in former times with significant larger effort.

The rest periods assigned at the start or end of an employee's shift is not tragic at all. The only consequences for the affected employees are that they may start or finish their work ten or twenty minutes earlier or later.

For these reasons we conclude that the solution produced by the break scheduling and task assignment tool is of high quality and the computational effort of five minutes to generate it is quite acceptable considering the complex nature of the problem.

Chapter 9

Conclusions

In this thesis we designed the domain specific language TEMPLE to model and solve staff scheduling problems. Thanks to TEMPLE, new software solutions for staff scheduling tasks can be obtained more quickly, and already existing solutions can be modified and extended more easily.

To approach the issue of staff scheduling we considered two real-life problems in Chapter 3 and Chapter 4. The first problem was a real-life break scheduling problem originating in the area of supervisory personnel. In this problem, breaks must be scheduled for a given shift plan in such a way that the obtained break schedules satisfy legal requirements and reduce shortage of staff to a minimum degree. To achieve that goal we developed a minimum-conflicts-based local search algorithm mimicking human experts when solving break scheduling problems. Computational results on real-life and randomly generated benchmark instances revealed that the minimum-conflicts-based algorithm can generate high-quality solutions that fulfill legal requirements and staffing demands at the same time.

The second real-life task was a related break scheduling problem originating from a call center. We adapted the min-conflicts-based algorithm to the additional and altered constraints of the call center problem, and that modified algorithm could again compute close-to-optimal solutions for real-life and randomly generated benchmark instances in acceptable time. The min-conflicts based algorithm is applied successfully at the call center where it is used to compute the daily break schedules for call center agents.

On the basis of the experience gathered from the two considered real-life tasks we abstracted common features and basic building blocks of staff scheduling problems and local search techniques in Chapter 5, and developed the domain specific language TEMPLE. In TEMPLE, a problem instance is modeled by small, concise building blocks reflecting common features of staff scheduling problems and local search techniques.

New building blocks are derived from already existing ones. By this principle a user is allowed to formulate a complex problem in small, concise and traceable steps. Consequently, the resulting problem models are well-structured, easy to understand, modify and maintain.

To transform TEMPLE models of staff scheduling problems into executable algorithms we developed and implemented a TEMPLE compiler in Chapter 7. The TEMPLE compiler translates a TEMPLE model into three local search algorithms, a simulated annealing algorithm, a hill-climbing based approach, and an iterated local search algorithm. Each of these algorithms can be executed instantaneously without requiring any further input from a user. To ensure that the obtained algorithms are carried out efficiently, we implemented several strategies within the compiler, in order that only as many computations as necessary are performed.

In Chapter 8 we delivered a proof of concept that real-life scheduling problems can be both effectively modeled and efficiently solved with TEMPLE. For that purpose we reconsidered the real-life break scheduling problem for supervisory personnel, and modeled it in TEMPLE. The resulting TEMPLE program was written in a very concise, understandable and modular manner, and consists of only 500 lines of code. Only one man-week was needed to develop the TEMPLE model for the break scheduling problem for supervisory personnel. Our experimental results on real-life and randomly generated benchmark instances revealed that with TEMPLE we could obtain solutions of acceptable quality at the same expenditure of time as the customized algorithm in Chapter 3.

Finally, in Chapter 8 we considered a multilayered break scheduling and task assignment problem. The goal for that staff scheduling problem was to develop a break schedule for a given shift plan and to assign tasks required to be performed by available employees in accordance with their qualifications. Again, many constraints were imposed on the break schedule as well as on the task assignment. Since the considered problem is very complex as a whole we decomposed it into three separate phases each of which modeled and solved by a separate TEMPLE program. In the first phase we computed a break schedule which is consistent with all legal requirements. In the second phase we optimized the break schedule with respect to the task requirements. In the third phase we assigned the required tasks to available employees and we optimized the task assignment with respect to the imposed criteria.

The three resulting TEMPLE models represent the core of a commercial break scheduling and task assignment tool. With a prototype of that tool we delivered a proof of concept that automated break scheduling and task assignment was possible within a reasonable amount of time, i.e., approximately five minutes on a state of the art computer. The prototype has been extended into a commercial application, which is already used successfully by decision makers in their day-to-day business.

For future work we want to model and solve further staff scheduling problems with our proposed domain specific language TEMPLE. In particular we want to address problems originating in areas of staff scheduling that we have not considered in this thesis, i.e., line of work construction and staff assignment. Moreover, we want to consider shift scheduling and break scheduling as a combined problem and solve it as a whole. By tackling these problems within one combined task we expect ourselves to be able to satisfy staffing requirements even more accurately. As a further topic for future work we want to make the TEMPLE modeling language even simpler in order that developers and end users can apply TEMPLE even more easily.

Appendix A

TEMPLE Model for the Break Scheduling Problem for Supervisory Personnel

A.1 General Settings and Constants

```
input                   = ".\2fc04a03.xml";
output                  = ".\sol-2fc04a03.xml";

algorithm               = iterated local search;

algorithm running time  = 60 minutes;
time slot               =  5 minutes;


int CYCLE_LENGTH                                        = 7 days;

//Constraint    C1 Break Positions
int MINIMUM_DISTANCE_TO_SHIFT_BORDER                    = 30 minutes;

//Constraint    C2 Lunch Breaks
int MINIMUM_DURATION_OF_LUNCH_BREAK                     = 30 minutes;
int MINIMUM_DISTANCE_LUNCH_BREAK_TO_SHIFT_START         = 3 hours 30 minutes;
int MAXIMUM_DISTANCE_LUNCH_BREAK_TO_SHIFT_START         = 6 hours;
int MINIMUM_DURATION_FOR_LUNCH_BREAK                    = 6 hours;

//Constraint    C3 Duration of Work Periods
int MINIMUM_DURATION_OF_WORKING_PERIOD                  = 30   minutes;
int MAXIMUM_DURATION_OF_WORKING_PERIOD                  = 100  minutes;

//Constraint    C4 Minimum Break Times after Work Periods
int CRITICAL_DURATION_OF_WORKING_PERIOD                    = 50 minutes;
int MINIMUM_BREAK_DURATION_AFTER_CRITICAL_WORKING_PERIODS  = 20 minutes;
```

```
//Break         C5 Duration
int MINIMUM_BREAK_DURATION                              = 10 minutes;
int MAXIMUM_BREAK_DURATION                              = 1  hour;


int VIOLATED                                            = 1;
int SATISFIED                                           = 0;

int INITIAL_BREAK_DURATION                              = 10 minutes;

int MAXIMUM_NUMBER_OF_BREAKS                            = 10;
```

A.2 Intervals and Links

```
Interval Problem;
Interval Requirement    with RequiredEmployees;

Interval Shift;
Interval Break;
Interval TimeSlot;

Problem         ->  Requirement;
Problem         <-> Shift;
Shift           <-> Break;

Problem         ->  TimeSlot;
Shift           <-> TimeSlot;
Break           <-> TimeSlot;
```

A.3 Constraint C_1 - Break Positions

```
Property Break::DistanceToShiftStart(Break thisBreak, Break.Shift[] associatedShift)
{
  DistanceToShiftStart = thisBreak.Start - associatedShift[1].Start;
}


Property Break::DistanceToShiftEnd(Break thisBreak, Break.Shift[] associatedShift)
{
  DistanceToShiftEnd = associatedShift[1].End - thisBreak.End;
}


HardConstraint Break::BreakPositions(Break thisBreak)
{
  if(thisBreak.DistanceToShiftStart < MINIMUM_DISTANCE_TO_SHIFT_BORDER)
    BreakPositions = VIOLATED;

  if(thisBreak.DistanceToShiftEnd < MINIMUM_DISTANCE_TO_SHIFT_BORDER)
    BreakPositions = VIOLATED;
}
```

A.4 Constraint C_2 - Lunch Breaks

```
Property Break::DistanceEndToShiftStart(Break thisBreak, Break.Shift[] associatedShift)
{
  DistanceEndToShiftStart = thisBreak.End - associatedShift[1].Start;
}


Property Break::IsLunchBreak(Break thisBreak)
{
  IsLunchBreak = true;

  if(thisBreak.Duration                 < MINIMUM_DURATION_OF_LUNCH_BREAK)
    IsLunchBreak = false;

  if(thisBreak.DistanceToShiftStart     < MINIMUM_DISTANCE_LUNCH_BREAK_TO_SHIFT_START)
    IsLunchBreak = false;

  if(thisBreak.DistanceEndToShiftStart  > MAXIMUM_DISTANCE_LUNCH_BREAK_TO_SHIFT_START)
    IsLunchBreak = false;
}


Property Shift::LunchBreakCount(Shift.Break[] scheduledBreak)
{
  LunchBreakCount = sum(i in scheduledBreak.getRange()) (scheduledBreak[i].IsLunchBreak);
}


HardConstraint Shift::LunchBreaks(Shift thisShift)
{
  if(thisShift.Duration > MINIMUM_DURATION_FOR_LUNCH_BREAK && thisShift.LunchBreakCount == 0)
    LunchBreaks = VIOLATED;
}
```

A.5 Constraint C_3 - Duration of Work Periods

```
Property Break::TimeInPosition(Break thisBreak, Break.Shift[] associatedShift,
                               Break.Shift().Break() scheduledBreak)
{
  TimeInPosition = thisBreak.Start - associatedShift[1].Start;

  selectMax(i in scheduledBreak.getRange() : scheduledBreak[i].End <= thisBreak.Start)
                                                                    (scheduledBreak[i].End)
  {
    TimeInPosition = thisBreak.Start - scheduledBreak[i].End;
  }
}


Property Break::TimeInBreak(Break thisBreak, Break.Shift().Break() scheduledBreak)
{
  TimeInBreak = thisBreak.Duration;

  select(i in scheduledBreak.getRange() : scheduledBreak[i].Start == thisBreak.End)
    TimeInBreak += scheduledBreak[i].Duration;
}
```

```
Property Shift::LastTimeInPosition(Shift thisShift, Shift.Break[] scheduledBreak)
{
  selectMax(i in scheduledBreak.getRange()) (scheduledBreak[i].End)
    LastTimeInPosition = thisShift.End - scheduledBreak[i].End;
}
```

A.6 Constraint C_4 - Minimum Break Times After Work Periods

```
HardConstraint Shift::MinimumBreakTimesAfterWorkPeriods(Shift.Break[] scheduledBreak)
{
  forall(i in scheduledBreak.getRange())
  {
    if(scheduledBreak[i].TimeInPosition > CRITICAL_DURATION_OF_WORKING_PERIOD)
      if(scheduledBreak[i].TimeInBreak  < MINIMUM_BREAK_DURATION_AFTER_CRITICAL_WORKING_PERIODS)
        MinimumBreakTimesAfterWorkPeriods = VIOLATED;
  }
}
```

A.7 Constraint C_5 - Minimum Break Durations

```
HardConstraint Break::BreakDurations(Break thisBreak)
{
  if(thisBreak.Duration < MINIMUM_BREAK_DURATION)
    BreakDurations = VIOLATED;

  if(thisBreak.Duration > MAXIMUM_BREAK_DURATION)
    BreakDurations = VIOLATED;
}
```

A.8 Constraint C_6 - Shortage of Employees

```
Curve Problem::StaffingRequirements(Problem.Requirement[] staffingRequirement)
{
  forall(i in staffingRequirement.getRange())
    StaffingRequirements.Pulse(staffingRequirement[i].Start,
                               staffingRequirement[i].End, staffingRequirement[i].Active,
                               staffingRequirement[i].RequiredEmployees);
}


Property Break::HasSuccessor(Break thisBreak, Break.Shift().Break() scheduledBreak)
{
  select(i in scheduledBreak.getRange() : scheduledBreak[i].Start == thisBreak.End)
    HasSuccessor = true;
}
```

```
Curve Shift::WorkingTime(Shift thisShift, Shift.Break[] scheduledBreak)
{
  WorkingTime.Pulse(thisShift.Start, thisShift.End, thisShift.Active);

  forall(i in scheduledBreak.getRange())
  {
    WorkingTime.Pulse(scheduledBreak[i].Start,
                      scheduledBreak[i].End,
                      scheduledBreak[i].Active, -1);

    if(scheduledBreak[i].HasSuccessor == false)
      WorkingTime.Pulse(scheduledBreak[i].End,
                        scheduledBreak[i].End + 1,
                        scheduledBreak[i].Active, -1);
  }
}


Curve Problem::WorkingTime(Problem.Shift[] scheduledShift)
{
  forall(i in scheduledShift.getRange())
    WorkingTime.Add(scheduledShift[i].WorkingTime);
}


Curve Problem::DeviationCurve(Problem thisProblem, Problem.Shift[] scheduledShift)
{
  forall(i in scheduledShift.getRange())
    DeviationCurve.CyclicAdd(scheduledShift[i].WorkingTime, CYCLE_LENGTH);

  DeviationCurve.Subtract(thisProblem.StaffingRequirements);
}


Curve Problem::ShortageCurve(Problem thisProblem)
{
  ShortageCurve.SubtractNegativeValues(thisProblem.DeviationCurve);
}


Property Problem::Shortage(Problem thisProblem)
{
  Curve shortageCurve = thisProblem.ShortageCurve;
  Shortage            = sum(i in shortageCurve.Period()) (shortageCurve.Value(i));
}


SoftConstraint Problem::ShortageOfEmployees(Problem thisProblem)     weight(10)
{
  ShortageOfEmployees = thisProblem.Shortage;
}
```

A.9 Constraint C_7 - Excess of Employees

```
Curve Problem::ExcessCurve(Problem thisProblem)
{
  ExcessCurve.AddPositiveValues(thisProblem.DeviationCurve);
}
```

```
Property Problem::Excess(Problem thisProblem)
{
  Curve excessCurve = thisProblem.ExcessCurve;
  Excess            = sum(i in excessCurve.Period()) (excessCurve.Value(i));
}


SoftConstraint Problem::ExcessOfEmployees(Problem thisProblem)       weight(2)
{
  ExcessOfEmployees = thisProblem.Excess;
}
```

A.10 Additional Constraints

```
Curve Shift::BreakPattern(Shift.Break[] scheduledBreak)
{
  forall(i in scheduledBreak.getRange())
    BreakPattern.Pulse(scheduledBreak[i].Start, scheduledBreak[i].End, scheduledBreak[i].Active);
}


HardConstraint Shift::NoOverlappingBreaks(Shift thisShift)
{
  Curve breakPattern = thisShift.BreakPattern;

  forall(i in breakPattern.Period())
    if(breakPattern.Value(i) > 1)
      NoOverlappingBreaks = VIOLATED;
}


HardConstraint Shift::ScheduleBreaksWithinShift(Shift thisShift, Shift.Break[] scheduledBreak)
{
  forall(i in scheduledBreak.getRange())
  {
    if(scheduledBreak[i].Start < thisShift.Start)
      ScheduleBreaksWithinShift = VIOLATED;

    if(scheduledBreak[i].End   > thisShift.End)
      ScheduleBreaksWithinShift = VIOLATED;
  }
}
```

A.11 Initialization

```
Property Shift::RequiredBreakTime(Shift thisShift)
{
  if(thisShift.Duration <= 10 hours)
    RequiredBreakTime = (int) (floor(((thisShift.Duration - 20 minutes) / 10.0)) * 2);
  else
    RequiredBreakTime = (int) (ceil(thisShift.Duration / 4.0));
}
```

```
Property Shift::NumberOfBreaks(Shift thisShift)
{
  int requiredBreakTime = thisShift.RequiredBreakTime;

  if(thisShift.Duration > MINIMUM_DURATION_FOR_LUNCH_BREAK)
  {
    NumberOfBreaks++;
    requiredBreakTime -= MINIMUM_DURATION_OF_LUNCH_BREAK;
  }

  NumberOfBreaks += (int) ceil (requiredBreakTime * 1.0 / MINIMUM_BREAK_DURATION);

  if(NumberOfBreaks > MAXIMUM_NUMBER_OF_BREAKS)
    NumberOfBreaks = MAXIMUM_NUMBER_OF_BREAKS;
}


Instantiate Shift.Break[] (Shift thisShift)
{
  Shift.Break[].Count     = thisShift.NumberOfBreaks;
}


Initialize Shift::BreakSchedule(Shift thisShift, Shift.Break[] scheduledBreak,
                                Shift.Problem[].TimeSlot[] timeSlot)
{
  int   numberOfBreaks = scheduledBreak.getRange().getUp();
  int[] breakStartTime;
  do
  {
    int requiredBreakTime = thisShift.RequiredBreakTime;
    int lunchBreakIndex   = -1;

    //Schedule a 30-minutes lunch break.
    if(thisShift.Duration > MINIMUM_DURATION_FOR_LUNCH_BREAK)
    {
      lunchBreakIndex = (int) floor((float) MINIMUM_DISTANCE_LUNCH_BREAK_TO_SHIFT_START /
                                    (float) (CRITICAL_DURATION_OF_WORKING_PERIOD + INITIAL_BREAK_DURATION));

      scheduledBreak[lunchBreakIndex].Duration  = MINIMUM_DURATION_OF_LUNCH_BREAK;
      scheduledBreak[lunchBreakIndex].Active    = true;
      requiredBreakTime                        -= MINIMUM_DURATION_OF_LUNCH_BREAK;
    }

    //Iterate over all other breaks and add 10 minutes to each break until entire break time is scheduled.
    int i = 1;
    while(requiredBreakTime >= INITIAL_BREAK_DURATION)
    {
      if(i > scheduledBreak.getRange().getUp()) i = 1;

      if(i != lunchBreakIndex)
      {
        scheduledBreak[i].Duration += MINIMUM_BREAK_DURATION;
        scheduledBreak[i].Active    = true;
        requiredBreakTime          -= MINIMUM_BREAK_DURATION;
      }
      i++;
    }
```

```
  if(requiredBreakTime > 0)
  {
    if(i > scheduledBreak.getRange().getUp()) i = 1;

    scheduledBreak[i].Duration += requiredBreakTime;
    scheduledBreak[i].Active    = true;
  }

  //Determine an initial legal break pattern by solving the corresponding STP.
  // + 2 because also shift start and shift end are variables of the STP.
  int numberOfSTPVariables  = numberOfBreaks + 2;

  STPSolver stpSolver = new STPSolver(numberOfSTPVariables);

  stpSolver.AddMinimumDistance(1, numberOfSTPVariables, thisShift.Duration);
  stpSolver.AddMaximumDistance(1, numberOfSTPVariables, thisShift.Duration);

  stpSolver.AddMinimumDistance(1, 2, MINIMUM_DISTANCE_TO_SHIFT_BORDER);

  if(scheduledBreak[1].Duration >= MINIMUM_BREAK_DURATION_AFTER_CRITICAL_WORKING_PERIODS)
    stpSolver.AddMaximumDistance(1, 2, MAXIMUM_DURATION_OF_WORKING_PERIOD);
  else
    stpSolver.AddMaximumDistance(1, 2, CRITICAL_DURATION_OF_WORKING_PERIOD);

  forall(i in 2..numberOfBreaks)
  {
    stpSolver.AddMinimumDistance(i, i+1, scheduledBreak[i-1].Duration +
                                         MINIMUM_DURATION_OF_WORKING_PERIOD );

    if(scheduledBreak[i].Duration >= MINIMUM_BREAK_DURATION_AFTER_CRITICAL_WORKING_PERIODS)
      stpSolver.AddMaximumDistance(i, i+1, scheduledBreak[i-1].Duration +
                                           MAXIMUM_DURATION_OF_WORKING_PERIOD);
    else
      stpSolver.AddMaximumDistance(i, i+1, scheduledBreak[i-1].Duration +
                                           CRITICAL_DURATION_OF_WORKING_PERIOD);
  }

  stpSolver.AddMinimumDistance(numberOfBreaks+1, numberOfBreaks+2,
                               MINIMUM_DISTANCE_TO_SHIFT_BORDER + scheduledBreak[numberOfBreaks].Duration);

  stpSolver.AddMaximumDistance(numberOfBreaks+1, numberOfBreaks+2,
                               MAXIMUM_DURATION_OF_WORKING_PERIOD + scheduledBreak[numberOfBreaks].Duration);

  if(thisShift.Duration > MINIMUM_DURATION_FOR_LUNCH_BREAK)
  {
    stpSolver.AddMinimumDistance(1, lunchBreakIndex + 1, MINIMUM_DISTANCE_LUNCH_BREAK_TO_SHIFT_START);
    stpSolver.AddMaximumDistance(1, lunchBreakIndex + 1, MAXIMUM_DISTANCE_LUNCH_BREAK_TO_SHIFT_START -
                                                         scheduledBreak[lunchBreakIndex].Duration );
  }

  breakStartTime = stpSolver.GetRandomSolution();
}
while(breakStartTime == null);

forall(i in 1..numberOfBreaks)
  scheduledBreak[i].Start = thisShift.Start + breakStartTime[i+1];

set{int} domainStart();
```

```
  forall(i in thisShift.Start .. thisShift.End)
    domainStart.insert(i);

  set{int} domainDuration ();
  forall(i in 0 .. MAXIMUM_BREAK_DURATION)
    domainDuration.insert(i);

  set{int} domainActive();
  forall(i in 0..1)
    domainActive.insert(i);

  forall(i in scheduledBreak.getRange())
  {
    scheduledBreak[i].Start.Domain.Add    (domainStart);
    scheduledBreak[i].Duration.Domain.Add (domainDuration);
    scheduledBreak[i].Active.Domain.Add   (domainActive);

    scheduledBreak[i].AddRelatedInterval(thisShift, "Shift");
  }

  forall(t in timeSlot[1].getRange())
  {
    if(thisShift.Start <= timeSlot[1][t].Start  && timeSlot[1][t].End <= thisShift.End)
    {
      timeSlot[1][t].AddRelatedInterval          (thisShift,        "Shift");
      thisShift.AddRelatedInterval               (timeSlot[1][t],   "TimeSlot");

      forall(i in scheduledBreak.getRange())
      {
        timeSlot[1][t].AddRelatedInterval           (scheduledBreak[i], "Break");
        scheduledBreak[i].AddRelatedInterval        (timeSlot[1][t],    "TimeSlot");
      }
    }
  }
}
```

A.12 Moves

```
Move Problem::BreakAssignment(Problem thisProblem, Problem.TimeSlot[] ts,
                              Problem.TimeSlot[].Shift[] scheduledShift)
{
  Curve shortage = thisProblem.ShortageCurve;

  select(i in ts.getRange() : shortage.Value(ts[i].Start) > 0)
    select(j in scheduledShift[i].getRange() : scheduledShift[i][j].WorkingTime.Value(ts[i].Start) == 0)
      scheduledShift[i][j].BreakAssignment;
}
```

```
Move Problem::BreakSwap(Problem thisProblem, Problem.TimeSlot[] ts,
                        Problem.TimeSlot[].Shift[] scheduledShift)
{
  Curve shortage = thisProblem.ShortageCurve;

  select(i in ts.getRange() : shortage.Value(ts[i].Start) > 0)
    select(j in scheduledShift[i].getRange() : scheduledShift[i][j].WorkingTime.Value(ts[i].Start) == 0)
      scheduledShift[i][j].BreakSwap;
}


Move Shift::BreakAssignment(Shift thisShift, Shift.Break[] scheduledBreak)
{
  range T = thisShift.Start .. thisShift.End;

  select(i in scheduledBreak.getRange())
    select(newPosition in T : newPosition != scheduledBreak[i].Start)
      scheduledBreak[i].Start = newPosition;
}


Move Shift::BreakSwap(Shift.Break[] scheduledBreak)
{
  select(firstBreak in scheduledBreak.getRange())
  {
    select(secondBreak in scheduledBreak.getRange())
    {
      int t                                 = scheduledBreak[firstBreak].Start;
      scheduledBreak[firstBreak].Start      = scheduledBreak[secondBreak].Start;
      scheduledBreak[secondBreak].Start     = t;
    }
  }
}
```

Appendix B

Break Scheduling and Task Assignment Tool

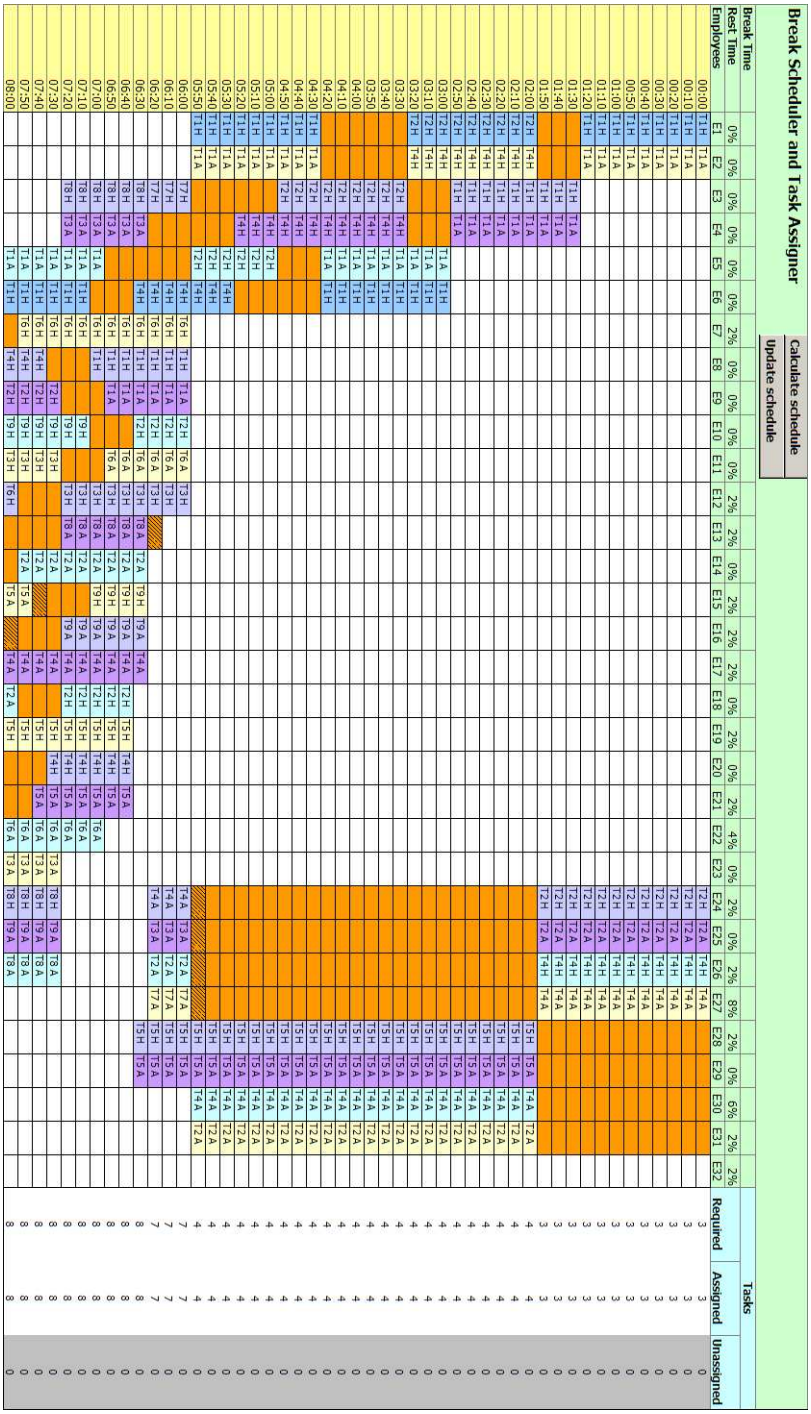


Figure B.1: Part one of the solution generated by our break scheduling and task assignment tool in Section 8.2.7.

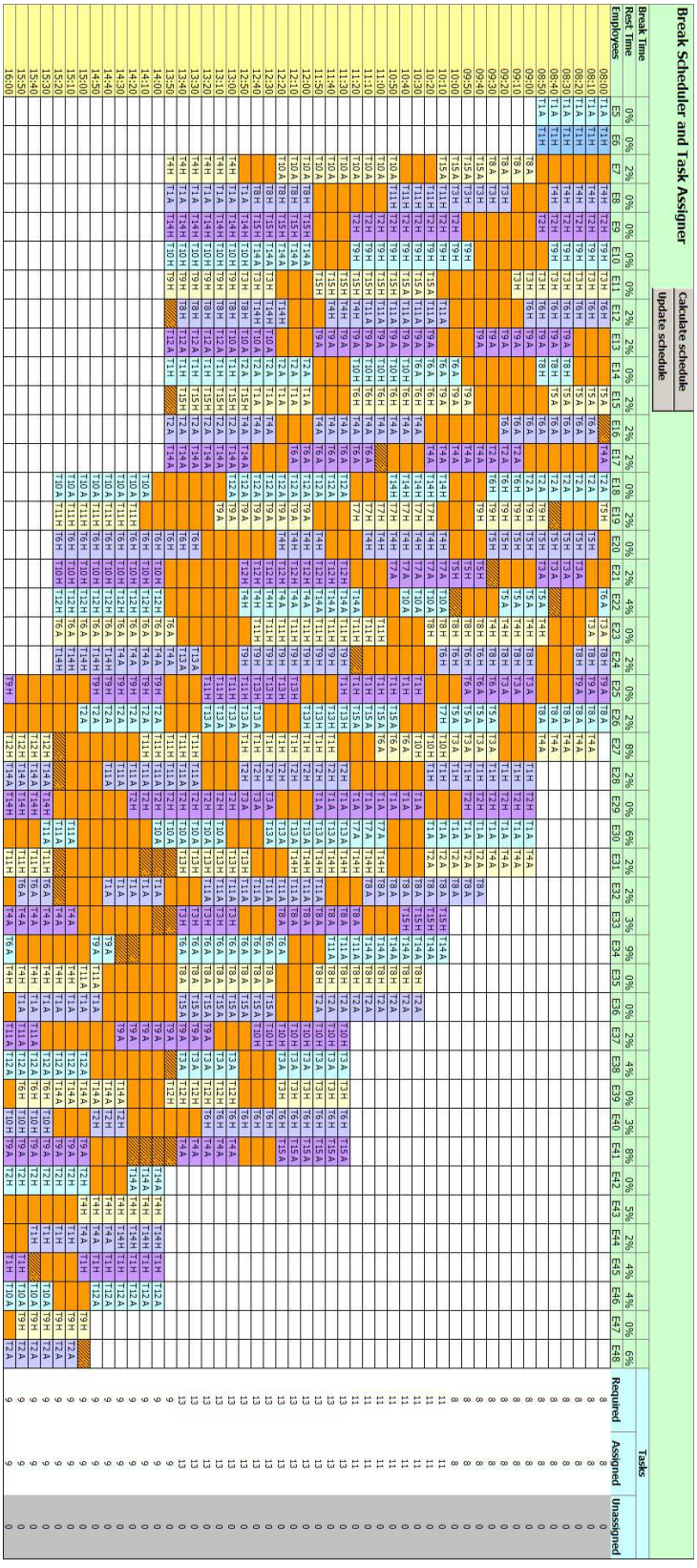


Figure B.2: Part two of the solution generated by our break scheduling and task assignment tool in Section 8.2.7.

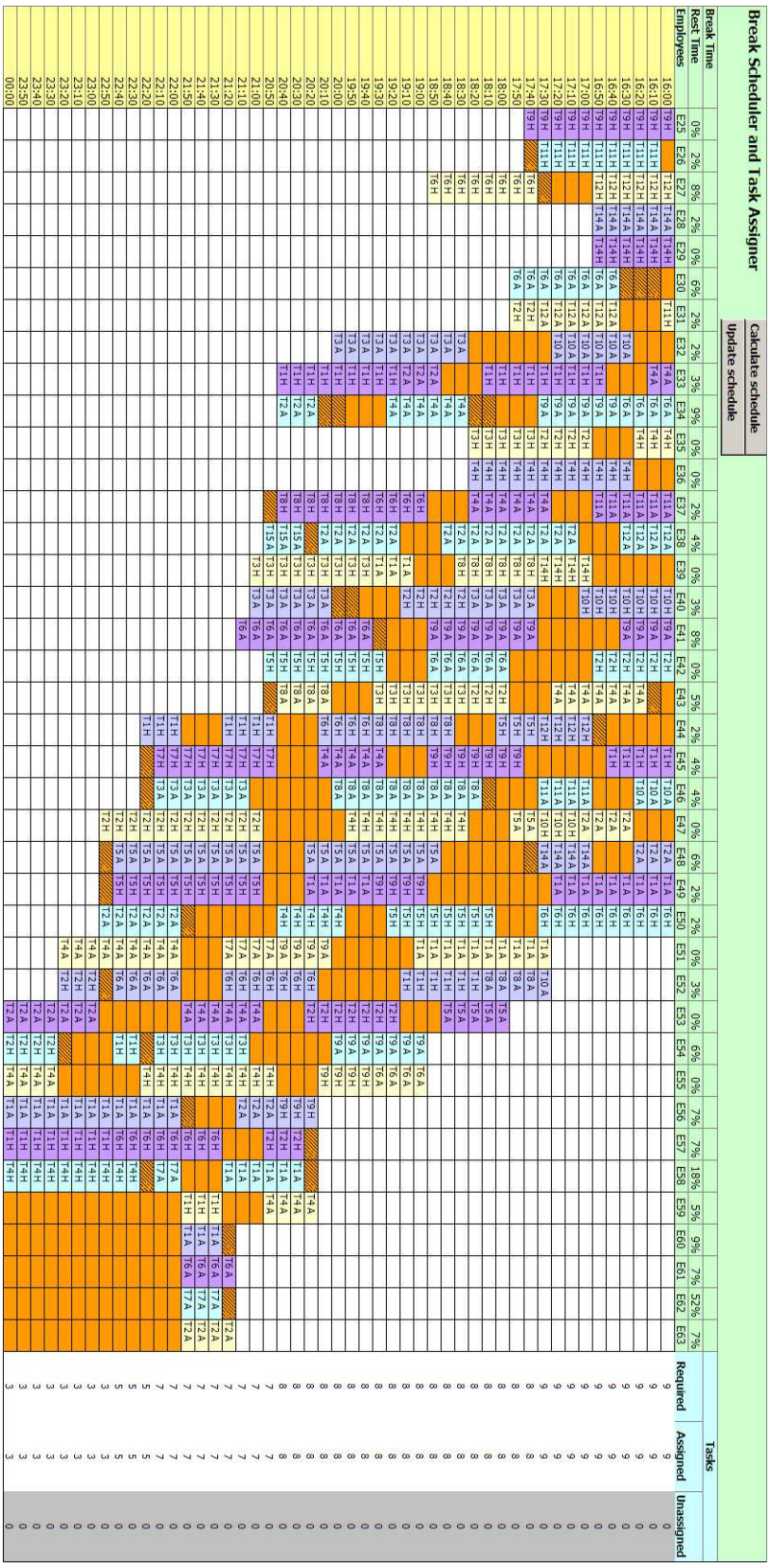


Figure B.3: Part three of the solution generated by our break scheduling and task assignment tool in Section 8.2.7.

Bibliography

[1] Zeynep Aksin, Mor Armony, and Vijay Mehrotra. The modern call center: A multi-disciplinary perspective on operations management research. *Production and Operations Management*, 16(6):665–688, 2007.

[2] J. P. Arabeyre, J. Fearnley, F. C. Steiger, and W. Teather. The airline crew scheduling problem: A survey. *Transportation Science*, 3(2):140–163, 1969.

[3] Turgut Aykin. Optimal shift scheduling with multiple break windows. *Management Science*, 42(4):591–602, 1996.

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

[5] Cynthia Barnhart, Amy Cohn, Ellis Johnson, Diego Klabjan, George Nemhauser, and Pamela Vance. *Handbook of Transportation Science, Airline Crew Scheduling*, volume 56 of *International Series in Operations Research and Management Science*. Springer, 2003.

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

[7] Jan Bisschop and Alexander Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, 20:1–29, 1982.

[8] Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The state of the art of nurse rostering. *Journal of Scheduling*, 7(6):441–499, 2004.

[9] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[10] Cyril Canon. Personnel scheduling in the call center industry. *4OR: A Quarterly Journal of Operations Research*, 5(1):89–92, 2007.

[11] Alberto Caprara, Matteo Fischetti, Pier Luigi Guida, Paolo Toth, and Daniele Vigo. *Solution of Large-Scale Railway Crew Planning Problems: The Italian Experience*. Springer, Berlin, 1999.

[12] Alberto Caprara, Matteo Fischetti, Paolo Toth, Daniele Vigo, and Pier Luigi Guida. Algorithms for railway crew management. *Mathematical Programming*, 79:125–141, 1997.

[13] Alberto Caprara, Paolo Toth, Daniele Vigo, and Matteo Fischetti. Modeling and solving the crew rostering problem. *Operations Research*, 46(6):820–830, 1998.

[14] Brenda Cheang, H. Li, Andrew Lim, and Brian Rodrigues. Nurse rostering problems - a bibliographic survey. *European Journal of Operational Research*, 151(3):447–460, 2003.

[15] George B. Dantzig. A comment on eddie's traffic delays at toll booths. *Operations Research*, 2:339–341, 1954.

[16] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[17] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988.

[18] Andreas T. Ernst, Houyuan Jiang, Mohan Krishnamoorthy, B. Owens, and David Sier. An annotated bibliography of personnel scheduling and rostering. *Annals of Operations Research*, 127:21–144, 2004.

[19] Andreas T. Ernst, Houyuan Jiang, Mohan Krishnamoorthy, and David Sier. Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*, 153(1):3–27, 2004.

[20] Andreas Fink and Stefan Voß. Hotframe: A heuristic optimization framework. In Stefan Voß and David L. Woodruff, editors, *Optimization Software Class Libraries*. Kluwer, 2002.

[21] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[22] Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. ESSENCE: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[23] Alex Fukunaga, Gregg Rabideau, Steve Chien, and Anita Govindjee. ASPEN: An application framework for automated planning and scheduling of spacecraft control and operations. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS97), Tokyo, Japan*, pages 181–187, 1997.

[24] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.

[25] Johannes Gärtner, Nysret Musliu, and Wolfgang Slany. A heuristic based system for generation of shifts with breaks. In *Proceedings of the Twenty-fourth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence (Springer) Cambridge*, 2004.

[26] Luca Di Gaspero, Johannes Gärtner, Guy Kortsarz, Nysret Musliu, Andrea Schaerf, and Wolfgang Slany. The minimum shift design problem. *Annals of Operations Research*, 155(1):79–105, 2007.

[27] Luca Di Gaspero and Andrea Schaerf. Easylocal++: An object-oriented framework for the flexible design of local-search algorithms. *Software - Practice and Experience*, 33(8):733–765, 2003.

[28] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 2006.

[29] Fred Glover and Manuel Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[30] Robert Harder. OpenTS - java tabu search framework. *http://www.coin-or.org/OpenTS/*, 2001.

[31] Pascal Van Hentenryck. *The OPL Optimization Language*. The MIT-Press, Cambridge, Mass., 1999.

[32] Pascal Van Hentenryck and Laurent Michel. *Contraint-Based Local Search*. The MIT Press, Cambridge, Mass., 2005.

[33] Martin Henz, Gert Smolka, and Jörg Würtz. Oz - a programming language for multi-agent systems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence, volume 1*, pages 404–409, 1993.

[34] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *ESEC / SIGSOFT FSE*, pages 62–73, 2001.

[35] Martin Stuart Jones. An object-oriented framework for the implementation of search techniques. Master's thesis, University of East Anglia, 2000.

[36] Scott Kirkpatrick, D. Gelatt Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[37] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Handbook of Metaheuristics*, chapter 11, pages 321–353. Springer, 2003.

[38] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[39] Zbigniew Michalewicz and David B. Fogel. *How to solve it: modern heuristics*. Springer-Verlag, 2000.

[40] Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1/2):43–84, 2000.

[41] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 83–100, 2002.

[42] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3):161–205, 1992.

[43] Nysret Musliu, Andrea Schaerf, and Wolfgang Slany. Local search for shift design. *European Journal of Operational Research*, 153(1):51–64, 2004.

[44] Nysret Musliu, Werner Schafhauser, and Magdalena Widl. A memetic algorithm for a break scheduling problem. In *Proceedings of MIC, VIII Metaheuristic International Conference, Hamburg*, 2009.

[45] Christos H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[46] Gregg Rabideau, Steve Chien, Alex Fukunaga, and Anita Govindjee. Iterative repair planning for spacecraft operations in the ASPEN system. In *Proceedings of the International Syposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS99), Noordwijk, The Netherlands*, 1999.

[47] Reza Rafeh, Maria J. García de la Banda, Kim Marriott, and Mark Wallace. From Zinc to design model. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, pages 215–229, 2007.

[48] Monia Rekik, Jean-François Cordeau, and François Soumis. Implicit shift scheduling with multiple breaks and work stretch duration restrictions. *Journal of Scheduling*, 13(1):49–75, 2010.

[49] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[50] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, 1993.

[51] Benjamin Smith, Rob Sherwood, Anita Govindjee, David Yan, Gregg Rabideau, Steve Chien, and Alex Fukunaga. Representing spacecraft mission planning knowledge in ASPEN. *AIPS-98 Workshop on Knowledge Engineering and Acquisition for Planning (AAAI Technical Report WS-98-03)*, pages 58–72, June 1998.

[52] Pascal Tellier and George White. Generating personnel schedules in an industrial setting using a tabu search algorithm. *E. K. Burke, H. Rudová (Eds.): PATAT 2006*, page 293Ű302, 2006.

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

[54] Gary M. Thompson and Madeleine E. Pullman. Scheduling workforce relief breaks in advance versus in real-time. *European Journal of Operational Research*, 181(1):139–155, 2007.

[55] M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.

[56] Mark Wallace, Stefano Novello, and Joachim Schimpf. Introducing ESRA, a relational language for modelling combinatorial problems. In *CP*, pages 214–232, 2003.

[57] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[58] Magdalena Widl. Memetic algorithms for break scheduling. Master's thesis, Vienna University of Technology, Austria, 2010.

[59] Magdalena Widl and Nysret Musliu. An improved memetic algorithm for break scheduling. In *Hybrid Metaheuristics*, pages 133–147, 2010.

Werner Schafhauser
Schelleingasse 17/6
A-1040 Wien

Persönliche Daten

Geburtsdatum	21. September 1981
Staatsbürgerschaft	Österreich
Zivildienst	abgeleistet
Sprachen	Deutsch (Muttersprache), Englisch (fließend) Italienisch (Grundkenntnisse)

Ausbildung

TU Wien

Okt. 2001 – Nov. 2010

Doktoratsstudium

8 Semester Doktoratsstudium der technischen Wissenschaften. Voraussichtlicher Abschluss: November 2010. Titel der Dissertation: *Temple - A Domain Specific Language for Modeling and Solving Real-Life Staff Scheduling Problems.*

Okt. 2006 – Nov. 2010

Masterstudium

4 Semester Masterstudium Computational Intelligence am 17. Oktober 2006 mit Auszeichnung abgeschlossen. Titel der Masterarbeit: *New Heuristic Methods for Tree Decompositions and Generalized Hypertree Decompositions.*

Nov. 2004 – Okt. 2006

Bakkalaureat

6 Semester Bakkalaureatsstudium Software & Information Engineering am 19. November 2004 mit Auszeichnung abgeschlossen.

Okt. 2001 – Nov. 2004

Gymnasium

8 Jahre Bundesgymnasium Villach, Peraustraße, Reifeprüfung am 26. Juni 2000 mit Auszeichnung bestanden.

Sept. 1992 - Juni 2000

Auszeichnungen

Juni 2007

Nominiert für den *Distinguished Young Alumnus/Alumna Award* für eine hervorragende Masterarbeit an der Fakultät für Informatik der TU Wien.

April 1998

Sieger der *Lateinolympiade*, Kärntner Landeswettbewerb, in der Gruppe 6. Klasse Langform.

Praktika und Berufserfahrung

TU Wien, Ximes

Projektassistent am Institut für Informationssysteme, Arbeitsgruppe für Datenbanken und Artificial Intelligence, und Entwickler bei Ximes GmbH, im Rahmen von Projekt Planning Knowledge des Kompetenz-Netzwerks Softnet Austria. Ziel des Projekts war die Entwicklung von Modellen, Methoden und Software-Lösungen für zeitbezogene Planungsaufgaben.

März 2007 – April 2010

Ximes

Praktikum als Entwickler. Programmierung von Algorithmen und statistischen Analyseverfahren für das webbasierte Time Intelligence Tool - Time Intelligence Solutions [TIS], in C# und ASP.NET.

Nov. 2006 – März. 2007

Infineon

Diverse Praktika als Entwickler.

- Programmierung von Erweiterungen der Infineon Standard Software Ceda/Cornerstone für die statistische Analyse von Produktionsdaten und deren grafischer Repräsentation.
- Entwicklung eines Reportingtools für die Qualitätsverbesserung von Produktionsdaten in MS Excel VBA.
- Implementierung von Perl/CGI-Scripts zwecks dynamischer Aktualisierung von Online-Projektdokumentationen.
- Programmierung eines Installations-, Setup- und Registrierungstools für Infineon Standard Software zur Extraktion und Analyse von Produktionsdaten.
- Evaluierung von IBM–MQSeries als Middleware für den Transfer von Produktionsdaten.

1998 - 2005

PUBLIKATIONEN

2010

A. Beer, J. Gärtner, N. Musliu, W. Schafhauser, and W. Slany. *An AI-based break-scheduling system for supervisory personnel*. IEEE Intelligent Systems, 25(2):60-73, 2010.

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*. HM 2010 – 7th International Workshop on Hybrid Metaheuristics. Lecture Notes in Computer Science, 2010.

2009

N. Musliu, W. Schafhauser, and M. Widl. *A Memetic Algorithm for a Break Scheduling Problem*. Proceedings of MIC – VIII Metaheuristic International Conference, Hamburg, 2009.

2008

A. Beer, J. Gärtner, N. Musliu, W. Schafhauser, and W. Slany. *Scheduling Breaks in Shift Plans for Call Centers*. In The 7th International Conference on the Practice and Theory of Automated Timetabling, Montral, Canada, 2008.

2007

N. Musliu and W. Schafhauser. *Genetic Algorithms for Generalised Hypertree Decompositions*. European Journal of Industrial Engineering, Vol. 1, No. 3, pp. 317-340, 2007.