Design and development of a programming language for constrained resource allocation

Design und Entwicklung einer Programmiersprache zur Ressourcenverteilung
Master-Thesis von Alina Dehler
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. rer. nat. Sebastian Erdweg

TECHNISCHE
UNIVERSITÄT
DARMSTADT

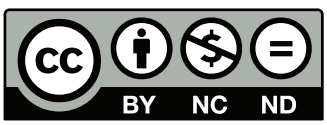
Department of Computer Science
Software Technology Group

Design and development of a programming language for constrained resource allocation
Design und Entwicklung einer Programmiersprache zur Ressourcenverteilung

Vorgelegte Master-Thesis von Alina Dehler

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. rer. nat. Sebastian Erdweg

Tag der Einreichung:



Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den May 3, 2016

(A. Dehler)

Abstract

Every year a school's administration has to assign the school's teachers to classes and lessons. In Germany this process is, compared to many other countries, very complex. Constraints for teacher allocation include official regulations, the teachers' and administration's wishes, and established best practices. However, there is no program or specific language available that can be used to compute a fair and suitable assignment.

In this thesis we therefore designed and implemented a domain specific language (DSL) that can model and solve the teacher allocation problem as a case study in language development for constrained resource allocation.

We gathered all relevant constraints that an assignment is subject to from domain experts of a local school. The language TCAL (teacher class assignment language) was developed with the Spoofax language workbench and utilizes the Choco solver. TCAL can represent all relevant data in a concise way and its syntax is easily readable and understandable. Using TCAL increases productivity compared to solving the problem with a general purpose language, as all required constraints and data constructs are already built in. With the data from a local school we can show that our language finds a feasible solution within approximately 30 seconds.

Contents

1 Introduction

Assigning tasks to workers, meetings to rooms, equipment to work sites or allocating channels in a wireless network are all examples of resource allocation problems. In this thesis we will focus on a specific resource allocation problem: the assignment of teachers to classes and lessons. To achieve an effective automatic assignment we designed and implemented a language that can describe a specific problem instance and solve it with the help of a constraint solver.

Assigning a school's teachers to lessons and creating timetables by hand is a rather tedious endeavor. Changing one small part of an assignment can trigger a chain reaction of inconsistencies that have to be fixed. In the past, a great number of hours had to be spent during the summer break to find a solution that seems as suitable and fair as possible.

While there are numerous programs that calculate the best possible timetable, we could not find any programs that automatically do the same for the assignment of teachers to lessons. One of the reasons might be that timetabling, unlike teacher assignment, is a more common problem around the world. Even though constraints might differ, the problem is largely the same. There are a number of lessons, that already have a teacher assigned, a number of time slots and a number of rooms. The goal is to assign every lesson to a time slot and room. Additional constraints must or should be met by the assignment. These constraints could be for example, that physics must be held in a physics room, or that Mrs. Schmidt should not have to teach before 9 am if possible, because her way to school takes over an hour.

The assignment of teachers to lessons is a far more diverse problem. Regulations are different in every country and even region, and depending on the people involved, the procedure may vary severely. In this thesis we focused on a problem definition that is common in Germany.

Students are grouped into classes and divided by grade and type of school (Hauptschule, Realschule and Gymnasium). The subjects that are taught in a class are fixed by the Ministry of Education in the form of hour boards. Teachers are usually trained to teach two subjects in either Gymnasium or Hauptschule and Realschule.

The school system in other countries, however, can deviate greatly from this scenario. In the United States and many other countries, students are not grouped into classes during high school, as they are in many European and Asian countries. For some lessons, students from different grades can even be mixed and there is no division into different school types, like Hauptschule, Realschule and Gymnasium. Students can choose a lot of their subjects, while the choice is very limited in Germany. Furthermore, teachers are usually assigned to courses before students are distributed to them. This is just one example to show how diverse this problem is and it could be the reason for the lack of software and research in this area that is applicable to a German school.

During the process of assigning teachers, many constraints must be considered. Of course a teacher must be qualified to teach a class and they must work a certain amount of hours per week. However, there are constraints that are more based on experience and intuition. Some Gymnasium teachers may be allowed to teach at Realschule and even though it is desired that some teachers keep their classes from the previous year, in other cases it may be imperative that the assigned teacher changes.

Herr Poprawa, who is one of the people responsible for the teacher allocation at the Otto-Hahn-Schule in Hanau, drew our attention to this issue. From him and the vice principal of the school Dr. Wolf we gathered all necessary requirements for the teacher allocation process. With these in mind we designed and implemented a language that can be used to model all relevant data in an expressive and unambiguous fashion, and specify what makes an assignment valid and what makes it better. The problem is then solved using constraint programming techniques.

1.1 Contribution

Within this thesis the domain-specific textual language TCAL (teacher class assignment language) is designed and implemented. As part of the realization, an intuitive grammar is introduced and with the help of the Spoofax language workbench the source code is parsed, desugared, analyzed and basic editor services are provided. As an intermediate step a Scala strategy is implemented to transform the resulting abstract syntax tree into custom Scala objects. To solve the underlying resource allocation problem, a model of the optimization problem is built and solved with the help of the Choco solver.

With exemplary data from a local school, we can successfully solve the teacher assignment problem. The language is extensible and the constraint solver can be easily replaced to improve the language in the future.

1.2 Outline

In Chapter 2 all relevant preliminaries are introduced. We explain what domain specific languages are and that language workbenches can greatly facilitate their design and implementation. We then introduce the language workbench Spoofax that was used to create TCAL. Subsequently we discuss the algorithmic problem that has to be solved in order to obtain a satisfactory assignment and we introduce the Choco library and solver that is used to solve the teacher assignment problem.

The requirements and the concept of the language implementation are presented in Chapter 3 of this thesis. By formulating the real world problems that teachers and principals have to solve, requirements and constraints for the language can be derived. With these in mind, the concept of TCAL is designed.

Chapter 4 depicts and describes TCAL's grammar. We explain the language constructs that are used to represent data, constraints and configurations in TCAL and explain their syntax.

To formalize the constraints that are relevant for the teacher assignment problem we give a mathematical representation of them in Chapter 5.

In Chapter 6 we discuss the implementation of TCAL. The syntax definition in SDF3, the name binding in NaBL and the definition of editor services are explained. We then discuss the Scala strategy that was designed to convert the abstract syntax tree of a TCAL program into Scala objects and then into a format appropriate for modeling with the Choco library.

We evaluate our language and its results in Chapter 7 by giving an overview of the language development process, the problems that we encountered on the way and the assignments that are created using our language.

In Chapter 8 related research is presented. This includes languages, data formats and frameworks that were created for similar problems and algorithms to solve the problem of teacher assignment or resource allocation.

Possible extensions and improvements of TCAL are listed in Chapter 9.

2 Preliminaries

In this chapter we will introduce the concept of domain specific languages (DSL) and language workbenches. We give an overview of the language workbench Spoofax and explain why we chose it to design and implement TCAL.

The underlying algorithmic problem of assigning teachers to classes and lessons is discussed in Section 2.2 and the Choco library and solver that is used for TCAL is introduced.

2.1 Domain Specific Languages and Language Workbenches

2.1.1 DSLs

The copious number of available programming languages can be divided based on their intended domain of use into general purpose languages (GPLs) and domain specific languages (DSLs). While GPLs, such as Java, Scala or C, are applicable in many different use cases, a domain specific language is a language that was developed with a specific application in mind. In contrast to GPLs, DSLs are primarily used to describe parts of a bigger software system [13]. Because the application of the language is known in advance, features and tasks that are commonly used in the domain can already be built-in features in DSLs. This can highly increase productivity. Furthermore the syntax can be more expressive and easier to understand and use for users that are less experienced in programming [26]. Because the notation in the language is usually at the same level as a domain expert's terminology, it can also greatly simplify communication with domain experts [13].

Some widely used DSLs are HTML for web development, SQL for managing data in a relational database, and shell scripts.

DSLs can either introduce their own grammar or they can be based on an already existing host language. The former is also called an external DSL [13]. Its grammar can be designed to accurately represent the domain's requirements. Depending on the application scenario the language must then be compiled or transformed. The latter, an internal DSL, reuses its host language's infrastructure and extends its syntax, but is thereby dependent on the host language's syntax and programming model [13].

One alternative to using or creating a DSL is to add a library to a GPL. The boundary between an internal DSL and a library or API is quite fuzzy. They can be best distinguished by the language nature. While for a library vocabulary is added for abstractions, internal DSLs add a whole grammar, where single constructs may only make sense in the context of a larger expression [13]. Even though libraries increase the expressiveness of a program written in a GPL and eliminate the amount of boilerplate code, they do not offer the advantages that a grammar designed specifically for the domain brings. In DSLs, domain-specific notations and expressive constructs are the basis when designing the language, in a library they have to be made fit into the GPL's syntax [26]. The programmer has to have basic knowledge of the GPL in order to use the library or an internal DSL.

DSLs vary in their degree of executability. DSLs can be executable and completely independent programming languages. Other DSLs are executable, but as input languages for other tools, like an application or parser generator, they have a more declarative nature. Still others are not meant to be executed and instead represent for example domain-specific data structures [26].

2.1.2 Language Workbenches

To support a programmer in the creation of a new language, a vast variety of tools are available. They include parser generators, meta-programming languages, frameworks, tools and frameworks for IDE

development or even tools that use the new language's specification to generate an entire integrated development environment (IDE) [18].

Language Workbenches combine these tools, that can support the developer in every step of the language development process, into one environment. They are tools that ease the creation and use of new languages and thereby increase a programmers productivity, as Fowler coined the term 2005 [12].

2.1.3 Spoofax

Spoofax is a language workbench for the development of textual software languages. The platform is available as a plug-in for the integrated development environment (IDE) Eclipse and as an Eclipse installation with Spoofax already preinstalled [11].

The Spoofax environment offers editor support for the meta DSLs that are used for language development and simultaneously for the developed language [18]. This way the developer can develop a language and use it within the same Eclipse instance. It is also possible to generate a stand-alone plug-in of the developed language for later use of the language.

While the developer is typing, syntax highlighting is added, errors and warnings are indicated, syntax and content completion is offered and the outline view is updated, to name just a few editor services. Error recovery is another important feature of Spoofax, that allows for editor services to work even in the presence of syntactic errors in the program [18].

SDF3 is a modular and declarative DSL that is used within Spoofax to define the syntax of a language. A language's grammar specifies its concrete and abstract syntax. Rules for basic editor services are automatically derived based on the language's grammar and saved as separate generated ESV files (editor descriptor language) [18]. They can be modified and extended by handwritten ESV files as needed.

Within a program, names can be used to reference program elements, such as variables, methods or objects. With the Name Binding Language NaBL the developer can specify name bindings and scoping rules to correctly bind a name's use site to its declaration during name analysis [25]. Semantic analysis, which includes name analysis and type analysis, is used for semantic editor services, code generation and consistency checking [18].

The Stratego transformation language is the default language for analysis, transformation and code generation in Spoofax.

From a user's language definition, Spoofax automatically derives a parser that runs in a background thread, desugares the parsed source file and analyzes it while the user is typing [18]. The result is an abstract syntax tree (AST) with various semantic information.

Spoofax enables a language development process without many of the difficulties that can come with developing a new language from scratch. It bans compatibility issues between different tools, and instead provides a single environment that automatically assists with or even takes over various tasks for the developer. Changes to one part of a language's definition take effect on all parts of the language and help the developer to maintain a consistent language definition. With these arguments in mind we chose to use Spoofax for the development of TCAL.

2.2 Constraint Optimization Problems and Solvers

The task of assigning teachers to classes and lessons is an algorithmic problem that can be formulated as a Constraint Optimization Problem. In this section, we explain the underlying algorithmic problem and give information about the strategy and the tool that we used to solve it.

2.2.1 General Problem Description

Resource Allocation

Resource allocation describes a very broad area of algorithmic problems. A limited resource must be assigned to competing activities, usually by optimizing an objective function that represents the quality of the assignment. Examples include:

- Allocation of channels in a wireless network
- Assigning flight crew members to flights
- Distribution of resources (manpower, tools etc.) to projects
- The nurse rostering problem (assignment of nurses to shifts)
- Construction of timetables for a school, university or examinations

Timetabling Problems

The creation of a timetable is a special case of the resource allocation problem. A general definition of the problem is the assignment of lessons to time slots in a week without overusing available resources. However, the problem can be specialized and adapted in many different ways. Schaerf [35] divided timetabling into three categories:

- **School timetabling:** creating a weekly school schedule, where teachers and classes can only be assigned once to the same time slot
- **Course timetabling:** creating a weekly university schedule, where lecturers can only be assigned once to a time slot while minimizing the amount of overlapping lectures for the student
- **Examination timetabling:** scheduling exams for courses to avoid overlap and spreading the exams over the examination time as much as possible for students

Specific problems can fall into one category or combine aspects of more than one.

Carter and Laporte [7] also include teacher assignment and classroom assignment as categories of the timetabling problem.

The creation of a satisfactory timetable is a problem that is encountered around the world at least once a year. Solving the timetabling problem by hand can take many days and the quality of the solution greatly depends on the person's experience. To improve the costs related to the creation of timetables and the quality of the result, automated timetabling has been subject to much research in the past decades. It is also an interesting research area for algorithmics, because the construction of timetables is NP-complete [10][9].

Teacher Assignment

The allocation of teachers to classes and lessons is a special case of the resource allocation problem, and depending on the definition, also of the timetabling problem.

Because the timetabling process is very different around the world, the starting situation for the assignment of teacher differs greatly. In the problem definition that we will focus on, the assignment of students to classes and lessons has already been fixed and the assignment to time slots and rooms will be done subsequently.

The objective of the teacher allocation is to assign a teacher to every lesson, while satisfying a number of constraints. Some constraints are compulsory (e.g. a teacher must be qualified to teach a subject), while others describe what makes an assignment better than another assignment (e.g. teachers have less overtime).

2.2.2 Constraint Satisfaction and Constraint Optimization Problems

Constraint Satisfaction Problems (CSP) can be modeled as a triple $< X, D, C >$, where X is a set of variables, D is a function mapping each variable in X to its domain (usually finite), and C a set of constraints [39]. A feasible solution to a CSP is found when a value within a variable's domain is assigned to every variable and all constraints are satisfied. Scheduling problems, where tasks have to be assigned to time slots, or the graph coloring problem, where in its simplest version no adjacent nodes can be assigned the same color, are examples for CSPs.

The eight queens puzzle is another popular example to explain CSPs. Eight queens have to be placed on a chessboard in a way that no queen threatens another. The problem has been extended to the general n-queens problem, where n queens have to be placed on a $n \times n$ chessboard.

One extension of the CSP is the Constraint Optimization Problem (CSOP). Tsang [39] defines a CSOP as a CSP with an additional optimization function. A CSOP can therefore be represented by $< X, D, C, f >$. The optimization function f maps every feasible solution of the CSP $< X, D, C >$ to a numerical value that has to be optimized.

When formulating constraints for a CSOP, they are either specified as hard or soft constraints. A solution is only feasible if no hard constraints are violated. A soft constraint should be obeyed, but may be broken if necessary. This will be penalized in the optimization function and decreases the quality of a solution.

Constraints can take several forms [34]. A constraint can be unary, which means that only one variable is effected by that one constraint by restricting its domain set. An example for a unary constraint is when planning a round trip through several cities, the last city can not be Darmstadt. The variable that represents the last stop of the round trip can not be assigned the value for Darmstadt. Binary constraints involve two variables. Constraints for the eight queens puzzle are usually binary, as they restrict the relation of the locations of any two queens. For binary constraints, the cross-product of the variable's domains is restricted. Higher-order constraints involve three or more variables. The *alldifferent* constraint is a common example. It enforces that no two values that are assigned to the effected variables can be the same.

CSPs and especially CSOPs are usually NP-complete [39].

2.2.3 Constraint Programming

Constraint Satisfaction Problems can be found not only in computer science, but in a vast variety of disciplines. To solve CSPs a new programming paradigm evolved from the artificial intelligence world in the sixties and seventies [33]. Instead of specifying a sequence of steps to solve the problem, as it would be done using imperative programming languages, a problem is modeled in a declarative manner and solved by a combination of constraint propagation and search strategies. This paradigm is called constraint programming (CP).

Modeling

When modeling a CSP or a CSOP it has to be identified what variables are necessary to cover the whole problem and what constraints have to be enforced to find a solution to the problem. The same problem may be modeled in different ways, where each model requires a different set of variables and constraints [34].

To model the eight queens puzzle the variables can be defined as the location of each queen. The domain of a variable represents all squares on the chessboard. The constraints can then be chosen to be binary constraints prohibiting that two queens are placed in the same row, column or diagonal.

Because we know that no two queens are allowed to be placed in the same row, there is another way to model the eight queens puzzle. There is one variable for each row and the value that must be

assigned represents the column in which a queen is placed within this row. The domain lies therefore in the number of columns. This model significantly decreases the search space.

Once the model is chosen, there are a number of languages available to define all decision variables, their associated domains, and declare the constraints on these variables. Some of the most common ones are logic programming languages (e.g. Prolog), modeling languages (e.g. MiniZinc or OPL) or a library within an imperative language (e.g. Choco or Gecode).

Solving

To solve a problem using the constraint programming paradigm, a CP solver is usually used. A CP solver is a constraint programming tool that often comes with its own language to model the problem and already has a variety of search strategies available. After defining the model and specifying the search strategy, a solver can take over the rest.

The runtime of solving a problem is highly dependent on the chosen strategy. The strategy can specify several aspects of the solving process. It usually specifies the search algorithm, constraint propagation, the heuristic used for selecting which variable to assign next, and which value of the domain to assign. Many search strategies include a combination of the following techniques [34]:

- **Backtracking Search** A decision variable is chosen and a value is assigned (depending on the specified heuristic). If the assignment does not violate any constraints, the next variable is chosen and assigned. If a constraint is violated the algorithm backtracks to a new instantiation. Figure 2.1 shows backtracking search for the four queens puzzle. A queen is placed in a row from left to right. If a constraint is violated, which is indicated by a red line, the algorithm backtracks.

- **Forward Checking** If a value is assigned to a variable, forward checking eliminates all values that become inconsistent from the domains of those variables that have not been assigned yet. However, only those variables are checked that are effected by a mutual constraint with the current variable.

- **Constraint propagation** By decreasing the size of a variable's domain (e.g. through forward checking or constraint propagation itself) other values can become inconsistent. These values are then also removed from the set of possible values. This step is repeated until there are no more changes. Unlike forward checking, not only constraints that involve the current variable are considered.

Using backtracking search alone is not always efficient. Figure 2.2 shows a situation in which it is clear that no solution can be found with the variables assigned as seen in the picture. No matter in which square the queen is placed in the last row, she is always threatened. When using only backtracking search, the algorithm will try to place the last queen in all four squares of the last row before relocating the third queen. In combination with forward checking, the domain of the last queen's position will be empty in the given situation and the algorithm can prune the search tree.

There are problems in which constraint propagation is enough to find a solution. Easy Sudoku puzzles are examples for this, when by progressively eliminating inconsistent numbers all squares are left with only one possibility.

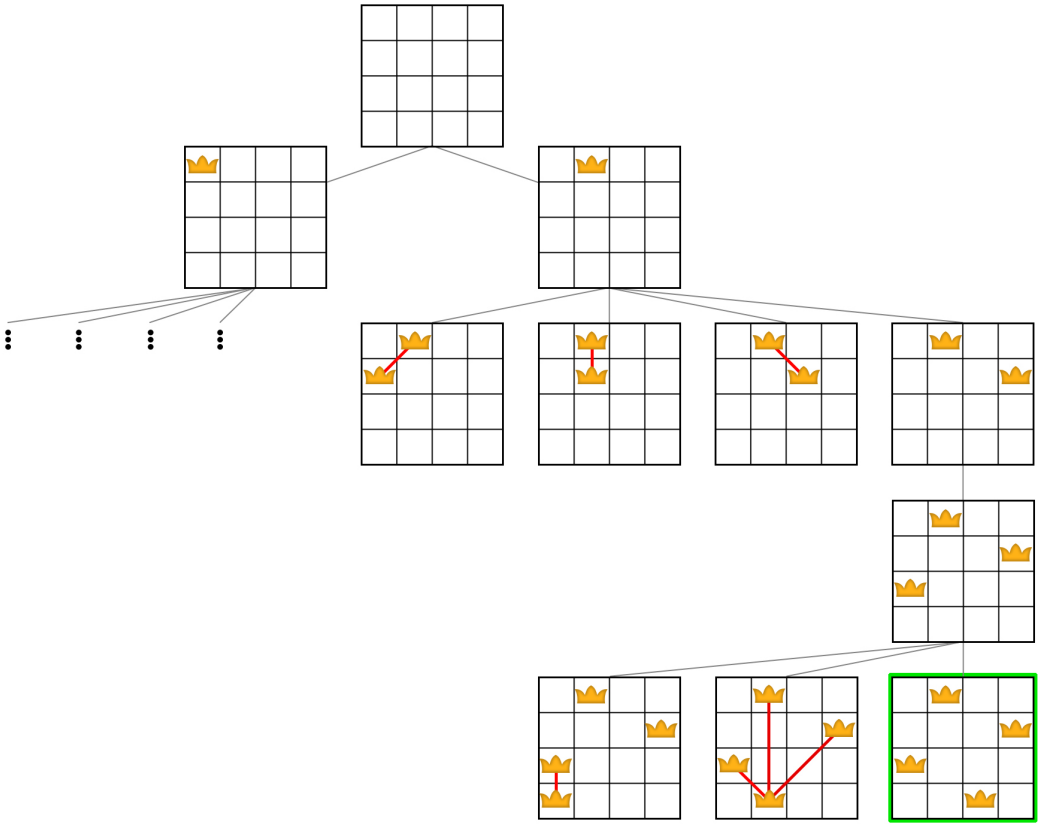


Figure 2.1: Solving the n-Queens Problem using backtracking

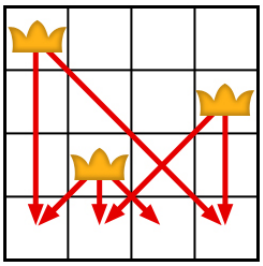


Figure 2.2: Example situation of inefficient backtracking search

Heuristics are valuable for effectively choosing the next variable and value to assign. They aim to keep the branching factor of the search tree as small as possible. The following are some general purpose heuristics regularly used [34]:

- **most-constrained-variable:** choosing the variable with the fewest possible values
- **most-constraining-variable:** choosing the variable that is involved in the most constraints
- **least-constraining-value:** assigning the value that eliminates the least values from other variables' domains

2.2.4 Choco

In order to solve the underlying resource allocation problem, the Java library Choco [30] is utilized.

The first version of the Choco solver was created by a French national initiative with the purpose of providing an open constraint solver, that can be used for both teaching and research [17]. It was initially written in the language CLAIRE and was later translated into Java. Choco is open source and distributed under BSD license. Because the source code can be found on GitHub, Choco is often used for teaching and research purposes. With its open API, users can implement their own algorithms and concepts. Choco is still being developed further. Its newest version 3.3.3 was released in December 2015.

To model problems in Choco, several types of variables are available: Integer variables, boolean variables, set variables consisting of integer values, real variables and expressions. The domain of values that a variable can attain must be defined beforehand. Additionally a wide variety of constraints can be used to model the problem at hand. Besides the usual arithmetical constraints, such as equality and inequality, logical constraints can be used to combine other constraints and global constraints simplify the programmer's work.

Choco's architecture is clearly separated into model and solver. After the user has modeled his problem, it is translated into a more CP-like model by the pre-processor and then automatically solved by the solver [17]. However, the user can also command the pre-processor to process constraints a certain way or use specific implementations. Advanced users can also change the solver's behavior through its API.

A variety of search strategies and heuristics are available in Choco to increase the resolution performance [30]. They mostly combine backtracking and constraint propagation.

A feasible solution to a modeled problem is found when the solver is able to assign a value to every variable from its domain while respecting all constraints over them.

Depending on the desired result of the solver's execution, the problem can be solved as a simple constraint satisfaction problem where the solver stops when the first feasible solution is found. The solver can also find all solutions or enumerate them. Furthermore, Choco can solve optimization problems. Here a objective variable must be declared and whether the variable is to be minimized or maximized. The solver can then find one or all optimal solutions.

There are several reasons why Choco was chosen for the implementation of TCAL. The Choco solver is still being actively developed further and it has an active community.

It is open source and as a Java library it can easily be integrated into the TCAL project.

Modeling problems with the Choco library is intuitive for Java developers with the help of the documentation and it enables expressive constraint modeling. Documentation and examples for standard use are available and good. Furthermore, Choco has won several medals at the MiniZinc challenge [37] and is among the fastest CP solvers on the market.

The biggest drawback that has been encountered during the implementation of TCAL with the Choco solver was the scarce documentation of how to best optimize a model and modify the solver's behavior.

3 Concept

To gather domain knowledge about the task of assigning teachers to lessons and classes, domain experts, namely Dr. Wolf and Herr Poprawa from the Otto-Hahn-Schule in Hanau, were consulted. They explained to us the regulations they have to follow during the process and the conventions they apply based on longtime experience. With this information the requirements were specified and a concept developed.

In Section 3.1 the problem at hand is described in natural language, with all the information that was given to us from the domain experts.

The choice of creating a DSL to describe and solve the teacher assignment problem is explained in Section 3.2. Here we define non-functional requirements for the language.

One of the main advantages of DSLs is the domain specific use of terminology and notations that make the language more expressive. In Section 3.3 the terminology that is used in TCAL is defined for a consistent understanding of its meaning. It is also shown what attributes for each data structure are necessary in order to express all relevant information for the task at hand.

3.1 Real World Problem Description

To design TCAL, the requirements of an exemplary school were used as a guideline. It has to be possible to express the following constraints in TCAL:

Hard Constraints

- For every class and subject there must be the necessary amount of teachers, in our case one teacher per lesson.
- A teacher must be qualified to teach the subject.
- A teacher can only teach in a class of his type (H,R or G), with the exception that a few teachers are allowed to teach all types.
- A teacher should teach the stipulated amount of periods per week. If positive or negative overtime is necessary, it can not exceed a specified amount.
- [Optional] If two subjects that are closely related are taught in the same class, the same teacher must be assigned to both (e.g. science course about biology and the subject biology).
- [Optional] The class teacher of a class should teach at least one subject in the class.

Soft Constraints

- Some teachers can teach lessons of another type, but it is less desirable.
- Overtime should be minimized.
- A specific assignment can be specified beforehand and should be met if possible. An assignment can be marked as desired or unwanted. Reasons for pre-assignments are:
  - Teacher taught the subject in the same class last year
  - The class teacher should teach as much as possible in the class
  - a teacher's wish
  - administrative decision

Additional conditions

- If a teacher teaches eight periods or more in grades 10 to 13, an additional period is credited due to more elaborate preparations. If a part time teacher teaches four periods or more in grades 10 to 13, an additional half period is credited. For partial positions, the credited periods must be calculated.

3.2 A DSL for Teacher Allocation

The assignment of teachers to classes is a complex process. Following regulations while keeping the teachers' and administration's wishes in mind and optimizing an assignment manually is a challenge.

We could not find a tool or language that was designed to help in the task of teacher allocation. We therefore decided to design and implement a DSL to support the school's administration with this endeavor.

As described in Section 2.1, by using the domain's terminology and an intuitive syntax, the DSL must be designed to be used by the school's administration, who may not have extensive programming experience. Even if the language is not used by the domain experts themselves, communication with them would benefit from an easily understandable notation when modeling the assignment problem.

Compared to modeling and solving the problem with an existing GPL, handling common constraints can be built-in to the DSL and a layer of abstraction simplifies its use and can increase productivity in the future.

For the design and implementation of the teacher class assignment language (TCAL) we set ourselves the following requirements:

- **Readable and understandable without programming experience**
  Domain terminology in combination with an intuitive syntax must be used for the implementation. This enables people with domain knowledge but without programming experience to read and understand a program written in TCAL with little or no instruction.

- **Unambiguous and expressive data representation**
  As we found out, the storage of the relevant data at a school can be unclear to anyone who is not directly involved in the teacher allocation process of the school. A precise and clear representation of the data will make maintenance easier and will be useful when communicating about it.

- **Minimize the amount of coding effort**
  Common tasks of the teacher allocation problem must be built-in. Constraints can be customized and added if required, but the realization must not be the duty of the user. The representation of the data and constraints must be concise.

- **Purely declarative modeling**
  Because the language must be usable for users with little programming knowledge, algorithmic details and the implementation of complex constraints and constructs must be hidden behind a layer of abstraction. For that reason the problem must be modeled in a declarative manner.

- **Language easily extensible**
  To achieve the goal of minimizing the amount of coding effort, a compromise has to be made with the power of expression. To introduce new constraints they have to be added to the language itself rather then giving the user the possibility to express arbitrary constraints. The language development project must facilitate easy extension of the language.

- **Efficient implementation**
  When describing a specific problem of realistic proportions and complexity with TCAL, the compiler and solver of TCAL should be able to solve it within a reasonable time with the capacities of a personal computer.

- **Problem solving tool effortlessly replaceable**
  To improve the efficiency of the implementation, it can be advantageous to try out and use different problem solving tools in the future. The structure of the language must enable an effortless replacement of the tool.

3.3 Definitions and Explanations of Terms

For a consistent use and understanding of important terms, we will explain in this section the terminology that is used in TCAL.

While the concept of a **teacher** is unambiguous, a **class** may have different meanings in different regions. We define a **class** to be a specific subgroup of students within one grade, for example class 5a.

In Germany schools are usually divided into **types**, namely Hauptschule, Realschule and Gymnasium. Every class is of one type. Teachers are trained to teach either Gymnasium or Hauptschule and Realschule, which is why we combined Hauptschule and Realschule into one type.

Classes can be combined into **couplings**, for instance if all students in one grade have Spanish together. To every class and coupling one or more **hour boards** are assigned. Usually all classes in the same grade with the same type are assigned the same hour board. An hour board contains information about how many hours in a week a **subject** (e.g. mathematics) is supposed to be taught. From this information a list of **lessons** can be constructed. Every lesson contains a list of classes, a subject and how many hours this subject should be taught. By running a program in TCAL the result is an **assignment** of teachers to lessons. Therefore an assignment is a result type and contains a teacher, a lesson and thereby a class and subject. Because certain assignments are specifically desired or unwanted, a **preassignment** can be formulated with a priority as part of the problem definition. To facilitate an incremental search, **incremental** preassignments can be specified. They are used to keep assignments from a prior execution of the program consistent in the next execution.

The following constructs are used to describe the data that is relevant for finding a satisfactory assignment of teachers to classes and lessons. If an attribute is enclosed in brackets it is optional. Objects that can be referenced from other program elements have a name (ID) to identify them.

Subject

Attribute	Description
ID	e.g. "mathematics"
[extended subject]	the subject this subject extends (e.g. advanced English extends English)

Type

Attribute	Description
ID	e.g. "HR" or "G" (representing Hauptschule/Realschule and Gymnasium)

Teacher

Attribute	Description
ID	e.g. "SmiA" for Adam Smith
name	the teacher's name
subjects	list of subjects that the teacher is qualified to teach
type	e.g. "HR", "G" or "all" type and whether teacher may teach another type if necessary
desired hours/week	number of periods that a teacher is supposed to work per week
[balance]	positive or negative amount of overtime from previous years
[extracurricular periods]	number of periods that the teacher needs for activities other than teaching, e.g. as a vice principal, administration

Class

Attribute	Description
ID	e.g. "G5a"
grade	e.g. "5"
type	e.g. "HR" or "G" type (representing Hauptschule/Realschule and Gymnasium)
hour boards	the class's subjects and the number of periods every subject is taught
class teacher	the class's class teacher, e.g. teacher "SmiA"

Coupling

Attribute	Description
ID	e.g. "G5abc"
classes	list of classes that are part of this coupling
hour boards	the coupling's subjects and the number of periods every subject is taught

Hour Board

Attribute	Description
ID	e.g. "G5"
(subject, hours/week)	a list of pairs containing the taught subjects and the mandatory number of periods that subject is taught, e.g. subjects that must be taught in all classes of grade 5 in Gymnasium

Preassignment

Attribute	Description
teacher	e.g. teacher "SmiA"
class or coupling	e.g. class "G5a"
subject	e.g. subject "mathematics"
wanted or unwanted	whether this assignment is desired or not e.g. if a teacher does not want to keep a class
priority	how important compliance with this preassignment is

Incremental Preassignment

Attribute	Description
teacher	one or more teachers e.g. teacher "SmiA"
class	one or more classes e.g. "G5a"
coupling	one or more couplings e.g. "G5abc"
subject	one or more subjects e.g. "mathematics"

For the incremental preassignment all attributes are optional, but at least one must be specified.

3.4 Data Basis and Output

To test our language with realistic data we received the hour boards and teacher data from the Otto-Hahn-Schule in the form of an Excel spreadsheet. To automatically convert the data into TCAL format we built a parser. However, we can not assume that data is stored in the same fashion at other schools. We therefore did not plan to add a built-in import option.

The resulting assignment should be saved in separate files:

Prettyprint

A nicely formatted output file must be created with two sections, yielding to two important points of view. The first section lists every teacher with their assigned lessons and further information. The second section is partitioned by classes and lists all lessons of that class with the assigned teachers.

Statistics

To help find errors in the model or to see the solver's statistics, an extra file must be printed. It shows the teachers' and lessons' intermediate data representation during compilation.

Raw Assignment

We want to offer the possibility for the user to keep parts of a previously computed assignment consistent in a new execution of the program. An assignment must therefore be saved in a format that can be parsed by the program in the next execution.

4 Language Syntax

TCAL is a domain-specific, declarative programming language for the task of assigning teachers to lessons and classes. The requirements we set for TCAL included the following:

a) readable and understandable without programming experience

b) unambiguous and expressive data representation

c) minimize the amount of coding effort

d) purely declarative modeling

To achieve those objectives, TCAL's keywords are domain specific terms and taken from the natural language that is used when talking about teacher allocation. Constructs for important elements, as described in Section 3.3, are available from the start and do not have to be defined by the user, as would be the case in a general purpose language. Algorithmic details are hidden behind a layer of abstraction and constraints can only be activated or customized, but do not have to be implemented.

A program in TCAL is an instance that is divided into three parts. The first part is made up of all the data elements. This includes preassignments and incremental preassignments.

In the second part of the program, constraints can be specified. The overtime constraint, specifying how much overtime is acceptable, is mandatory. Other constraints, like the type constraints, that allow some teachers of one type to teach another type, are optional.

The last part of a program contains the configuration specifications. A time limit can be set here and several output files can be generated after an assignment was computed.

4.1 Common Language Elements

Comments

We follow a common style for comments. A double slash // starts a comment until the end of the line and /* starts a multi-line comment, while */ closes it.

Identifiers

Identifiers are used to reference other program elements, such as teachers, subject, hour boards or classes. An identifier must start with a letter and can contain letters, numbers, underscores and hyphens.

4.2 Data

The definition of a data object always begins with a keyword referring to the data type, namely instance, type, subject, teacher, hourboard, class, coupling, preassignment or incremental. If the data object can be referenced from another object, an ID follows. Most data objects (type and subject are the exception) have several attributes that the user must specify. They are defined within curly brackets by stating the attribute, followed by a colon : and the user's value.

Except for the instance declaration, the order in which data objects are defined can be decided by the user. Having the declaration of the hour board for all classes in one grade close to those classes will simplify maintenance for later years. Hour boards that are assigned to different grades and types can be specified in a separate section for instance.

Many objects in TCAL reference other objects. The order in which objects are referenced is important for compilation (more details in Chapter 6 and Figure 6.3), but when writing a program in TCAL the order of the data objects can be chosen by the user. Nevertheless we will introduce the data constructs in order of compilation to introduce each before referencing it.

Instance

At the beginning of every TCAL program, the instance has to be given a name. Names are only bound within the current instance and several instances representing the same school but different years, can be saved within the same project.

Type

Every class has a type, in Germany the types are Gymnasium, Realschule and Hauptschule. From an administrative standpoint, Realschule and Hauptschule can be combined. This results in the types *HR* and *G* for our exemplary school. Teachers are usually only qualified to teach one type.

Subject

In most cases a subject only consists of its name, that can be referenced from other objects. Two subjects are already built in, *homeroom* and *coordination*. Homeroom is a lesson that is always taught by the class teacher. Problems in the class can be discussed, projects and excursions planned. In the coordination lesson some of a class's teachers meet without students.

In some cases subjects are taught for which no teacher is specifically qualified, but they can be taught by a teacher that is qualified in a related subject. As seen in Code Listing 4.1, teachers that are qualified to teach *English* are also allowed to teach *advancedEnglish*. The lesson *LQ (Lions-Quest)* is moderated by the class teacher and therefore extends homeroom.

When an extending and the extended subject is taught in the same class the teacher of both should be the same.

```
1 instance summer16
2
3 type HR
4 type G
5
6 subject mathematics
7 subject English
8 subject advancedEnglish extends English
9 subject LQ extends homeroom
```

Code Listing 4.1: Definition of types and subjects in TCAL

Teacher

Every teacher has a name and an ID that he can be referenced by. In Code Listing 4.2, Adam Smith can be referenced with *SmiA*. The subjects a teacher is qualified to teach are references to subject objects, the same applies to the teacher's type. When the type is followed by a +, the teacher is allowed to teach other types if an applicable constraint is specified in the constraint section of the program (more in Section 4.3). Additionally the desired hours are specified and optionally the balance of previous overtimes and the hours a teacher spends for activities other than teaching a class. If wanted, the extracurricular activity can have a label to describe it. For multiple extracurricular activities the hours should be summed up and the label can contain multiple activities.

```
teacher SmiA {
  name: "Smith, Adam"
  subjects: mathematics, english
  type : G+
  hours: 25
  balance : 1.75
  extracurricular : 2 "library, project"
}
```

Code Listing 4.2: Definition of a teacher

Hour Board

For every grade of one type there is usually an hour board defined by the government. Instead of copying the same subjects and hours to every class, one hour board can be assigned to them all. Additionally, subjects like religion or languages are not taken by all students and are not taught in all classes. By combining hour boards all needed combinations can be achieved.

Every hour board has an ID that can be referenced from other objects. It contains a variable number of subject references with the amount of hours this subject must be taught.

One exceptional case is the *coordination* lesson. The number of teachers is variable here, only the class teacher is always involved. In Code Listing 4.3 the teachers that are assigned to teach *mathematics* and *German* in the same class must also be assigned to the *coordination* lesson.

```
hourboard G5 {
    mathematics : 4
    German : 5
    history : 2
    coordination mathematics, German : 2
    homeroom : 2
}

hourboard G5Languages {
    Spanish : 2
    French : 2
}
```

Code Listing 4.3: Definition of hour boards

Class

Every class has an ID, a type, a grade and a reference to its class teacher. The class teacher is automatically assigned to possible *homeroom* lessons and lessons that extend *homeroom*, as well as the *coordination* hour. One or more hour boards are assigned to every class. With this information, a list of lessons can be constructed for further computations (more in Section 6.4).

```
class G5a {
    type : G
    grade : 5
    classteacher : SmiA
    hourboard : G5, G5Sport
}
```

Code Listing 4.4: Definition of a class

Coupling

In some cases not all students of one class take the same subject and as a result students of different classes are combined. Students can for instance choose to attend protestant religion, catholic religion or ethics. The same applies for languages. A coupling contains references to all classes that are combined for the subjects in the referenced hour boards.

```
coupling G7abc {
    classes : G7a, G7b, G7c
    hourboard : G5Languages
}
```

Code Listing 4.5: Definition of a coupling

Preassignment

In the description of the soft constraints in Section 3.1 we gave a number of reasons why certain assignments are wanted or unwanted prior to executing the program. These can be specified in a preassignment object, containing references to the teacher, the class and the subject. It must also be stated if the assignment is *wanted* or *unwanted*. The priority is added as penalty to the optimization function in case the preassignment is not obeyed in the resulting assignment. Satisfying a preassignment with a higher priority value is therefore more important and increases the quality of a solution. The priority can also be set to *must* if the preassignment should be treated as a hard constraint.

```
preassignment {
    teacher : SmiA
    class : G5a
    subject : mathematics
    wanted
    priority : 10
}
```

Code Listing 4.6: Definition of a preassignment

Incremental Preassignment

If the output file of a previous execution of the program is set as input for incremental preassignment in the program's configuration section (see Section 4.4), incremental preassignment objects are used to create mandatory preassignments. An incremental preassignment can contain references to subjects, classes, couplings and teachers. All lessons are then filtered for those that contain any combination of the specified references. A mandatory preassignment is created for all filtered lessons.

Code Listing 4.7 specifies that all mathematics lessons that were assigned to teachers SmiA or LarK in the previous solution get assigned to the same teachers in the next execution of the program.

```
incremental {
    subjects : mathematics
    teacher : SmiA, LarK
}
```

Code Listing 4.7: Definition of an incremental preassignment

4.3 Constraints

Some constraints are already built-in, but there are a few that have to be explicitly specified.

The optimization function in TCAL minimizes the score of a solution. When a soft constraint is not obeyed, a penalty is added to the score. The maximum penalty that can currently be specified is 100. As mentioned before, not effectuating a preassignment adds a penalty to the score. The violation of other soft constraints can also add penalties. In this section of the program, the user can specify them.

The overtime constraint must be specified in a TCAL program. The specified range in Code Listing 4.8 line 2 declares that a teacher's hours per week can be at most 4 less than the desired hours and at most 2 more.

Because it is most desirable for teachers to have no overtime, every deviating hour is penalized. The default penalty is 1, but with the optional specification *=> 2*, the penalty for every hour of overtime is set to 2.

The constraint of adding the teachers' balance to the overtime constraint, as in line 3, is only relevant for teachers that have overtime from the previous year. With the teacher's balance added to his desired hours, the scheduled hours can not exceed the boundaries set in this second range. This can be best explained with an example:

> *A teacher's desired hours are 26. He has a balance of -5 from the previous year. Without adding the balance constraint, the teacher would be able to work 22 to 28 hours, which could very well result in increasing the teacher's overtime. Adding the teacher's balance would result in an allowed span of 27 to 33 hours. This is not reasonable, as the desired hours are not even within the range anymore. When taking the second range ([-5,3]) into account, the hours can not exceed 26+3=29 and therefore stay in a more reasonable realm with 23 to 29 hours.*

At the end of an overtime constraint, an option can be specified. One possible option is *soft*. As a result the given overtime bounds are not used to create hard constraints. Instead, for teachers that exceed these bounds, the overtime penalty is computed and multiplied by 5. If no solution can be found with hard constraints, this option gives the possibility to find out which assignments, subjects or teacher are causing the problem. However, this has a very negative effect on the resolution performance.

The second available option is *quadratic*. The amount of hours that the scheduled hours deviate from the desired hours are squared before multiplying them with the overtime penalty. This way a uniform distribution of overtime over all teachers is more advantageous.

Another kind of constraint that can be set in this section is the type constraint (lines 4 and 5). Usually a teacher's type must match the class's that he is assigned to teach. A teacher can however have a "plus type". In this case a teacher with type *G+* can for instance teach classes of type *HR*, as seen in Code Listing 4.8 line 4. If this constraint should only be applied to certain grades, a range can be specified. Line 5 therefore states that teachers of type *HR+* can teach classes of type *G* only in grades 5 to 10 and with a penalty of 3. If no penalty is set, a teacher teaching his "plus type" is not penalized.

If the constraint *add classteacher lesson* is set, as in line 6, every class teacher must be assigned to at least one lesson in his class.

The constraint in line 7 specifies that a teacher who teaches eight or more hours in the upper classes (grades 10 to 13), is credited one additional hour.

```
constraints {
    overtime [-4,2] => 2 quadratic
    add balance [-5,3]
    G can teach HR => 2
    HR can teach G [5,10] => 3
    add classteacher lesson
    add 1 hour when 8 hours in upper
}
```

Code Listing 4.8: Constraints in TCAL

4.4 Configuration

The last part of a TCAL program is the optional configuration section.

In this section, the time limit of the solver can be set. In Code Listing 4.9 line 2 the time limit is set to 180 seconds. If no time limit is set, the solver will search until the optimal solution is found. This is not recommended for bigger problem instances.

The next three configurations in the Code Listing 4.9 determine what files will be created if a solution is found and what the file names will be. In the example a nicely formatted prettyprint will be written into a file called *UV_pretty.txt*. It includes a table with a row for every teacher with a teacher's hours per week, his overtime and what lessons he will teach. Subsequently, a table for every class lists all of the class's lessons and the assigned teachers, including lessons of couplings that involve the class. The file that will be created when the statistics configuration is set is more interesting for troubleshooting. It contains the arrays for teachers and lessons that are created during compilation and are used as input for the solver. Additionally the lists of penalties and the solver's statistics are saved here.

A file containing a triple of class/coupling, subject and teacher for every assignment is always created when a solution is found. If the *incremental* configuration is not set, this file is saved as *output.txt*. If a solution is not completely satisfactory to the user, the file with the name specified after *incremental* can be used in combination with incremental preassignments (see 4.2) to preserve assignments during the next execution of the program for an incremental search.

```
configuration {
    timelimit 180
    output "UV_pretty"
    statistics "UV_stats"
    incremental "UV_output"
}
```

Code Listing 4.9: Configurations in TCAL

5 Encoding as constraints

To formalize and disambiguate the constraints listed in Section 3.1, mathematical representations of all constraints are given in this chapter. These constraints are thereafter used to implement the constraint model with Choco in Section 6.5.

Input

P the set of all types (e.g. "G" or "HR")

S the set of all subjects

T the set of all teachers, $\forall t \in T$:

- $s[t] \subseteq S$ is the set of subjects the teacher can teach
- $h[t] \in \mathbb{R}$ is the amount of hours the teacher should teach per week
- $t[t] \in P$ is the type the teacher can teach

L the set of all lessons, $\forall l \in L$:

- $s[l] \in S$ is the subject of the lesson
- $h[l] \in \mathbb{R}$ is the amount of hours this lesson is taught per week
- $c[l] \in T$ is the class teacher of lesson's class
- $g[l] \in \mathbb{N}_0$ is the grade of the lesson's class
- $t[l] \in P$ is the type of the lesson's class
- $i[l] \subseteq L$ is the set of linked lessons

C the set of all classes

Y the set of all type constraints of form (t_1, t_2, g_1, g_2, p) representing that a teacher of type $t_1 \in P$ can teach in a class of type $t_2 \in P$ in grade $g_1 \in \mathbb{N}_0$ to $g_2 \in \mathbb{N}_0$ with $g_1 < g_2$ and penalty $p \in \mathbb{N}_0$ (for the mathematical representation we will ignore the option where no bounds are specified)

A the set of all preassignments of form (t, l, w, p) representing that teacher $t \in T$ teaching lesson $l \in L$ is wanted if w is 1 and unwanted otherwise with penalty $p \in \mathbb{N}_0$ if preassignment is not realized or a mandatory assignment if $p = 9999$ ("must")

Decision variables

- for each teacher $t \in T$ and each lesson $l \in L$ a variable $x[t, l]$ is introduced with domain 0,1
- $x[t, l] = 1$ means that teacher t is assigned to lesson l

Penalties

for all $t \in T$ there is $p'_t \in \mathbb{N}_0$, the penalty for this teacher's overtime
P'_T the set of all overtime penalties
for all $a \in A$ there is $p''_a \in \mathbb{N}_0$, the penalty for this preassignment
P''_A the set of all preassignment penalties
for all $l \in L$ there is $p'''_l \in \mathbb{N}_0$, the penalty if the type of the assigned teacher is not the type of the lesson's class
P'''_A the set of all type penalties

Objective

Minimize the score of the assignment (the sum of all penalties).

Hard Constraints

Every lesson is assigned to exactly one teacher (except for coordination hour)

$$\forall l \in L : \sum_{t \in T} x[t,l] = 1$$

A teacher must be qualified to teach the lesson's subject

$$\forall t \in T, \forall l \in L : x[t,l] = 1 \implies s[l] \in s[t]$$

Home room must be taught by the class teacher

$$\forall t \in T, \forall l \in L : s[l] = homeroom \wedge c[l] = t \implies x[t,l] = 1$$

The class teacher teaches at least one lesson in his class (optional constraint)

$L'_{t,c} \subset L$ the set of lessons that the class teacher t of class c is qualified to teach

$$\forall c \in C : \sum_{l \in L'_{c[l],c}} x[c[l],l] > 0$$

If a link to other lessons exists the same teacher has to be assigned to all

$$\forall t \in T, \forall l \in L : |i[l]| > 0 \wedge x[t,l] = 1 \implies \forall l' \in i[l] : x[t,l'] = 1$$

Mixture of Hard and Soft Constraints

A teacher can only teach in a class of his type (HR or G). If he has type "all" he can teach all types.

$$\forall t \in T, \forall l \in L : x[t,l] = 1 \implies (t[t] = t[l] \vee t[t] = all \vee$$
$$\exists g_1, g_2, p \in \mathbb{N}_0 (g_1 \leq g[l] \wedge g[l] \leq g_2 \wedge p'''_l = p \wedge (t[t], t[l], g_1, g_2, p) \in Y)$$

A teacher should teach the stipulated amount of periods per week. Depending on the teacher's overtime balance from last year, his extracurricular hours and the allowed range that is specified, a lower and upper bound is calculated. The overtime is used as penalty.

$$\forall t \in T : \sum_{l \in L} (x[t,l] * h[l]) \leq h[t] + upper \wedge \sum_{l \in L} (x[t,l] * h[l]) \geq h[t] - lower \wedge {p_t}' =$$
$$|h[t] - \sum_{l \in L} (x[t,l] * h[l])|$$

Soft Constraints

A preassignment should be realised if possible, otherwise a penalty is added.

$$\forall (t,l,w,p) \in A : (w = 1 \implies (x[t,l] = 1 \oplus ({p_{(t,l,w,p)}}'' = p \wedge p \neq 9999)) \vee (w = 0 \implies (x[t,l] =$$
$$0 \oplus ({p_{(t,l,w,p)}}'' = p \wedge p \neq 9999)))$$

Objective Function

The goal of the assignment is to find a feasible solution with the following objective function:

$$minimize(\sum_{t \in T} {p_t}' + \sum_{a \in A} {p_a}'' + \sum_{l \in L} {p_l}''')$$

6 Language implementation

During the implementation of a DSL, it is good practice to incrementally add new features to the language and refine its characteristics as needed [40].

When introducing a new feature, it has to be considered what additions or modifications of the language have to be made. Figure 6.1 shows all aspects of TCAL that have to be considered.

Within the Spoofax framework the syntax is defined using the syntax definition formalism SDF3. In Section 6.1 the realization of the syntax definition is further discussed.

When trying to refer to elements within a program, such as variables or methods, name resolution algorithms are necessary to establish the appropriate name binding. To link use sites of an element to its definition, name binding and scoping rules are used. When adding a new feature to TCAL, we therefore have to make sure that we add the appropriate name binding and scoping rules to the language or modify the existing ones. They are defined in Spoofax with the Name Binding Language NaBL. NaBL is a declarative domain-specific language, from which an efficient name resolution algorithm is then derived [25]. In Section 6.2 we discuss the name binding of TCAL further.

Basic editor services are automatically derived based on the language's grammar by Spoofax. We added minor changes to the syntax highlighting and outline view, which is discussed in Section 6.3.

Within Spoofax's architecture the input of a TCAL file is parsed, desugared and analyzed, which results in an abstract syntax tree (AST) of Stratego terms [18]. Instead of using Spoofax's transformation language Stratego, we use Scala and pattern matching in a custom Scala strategy to transform the AST into Scala objects (Section 6.4.1).

We chose to use the Choco solver library to solve our underlying constraint optimization problem. In Section 6.4.2 we discuss the conversion of the Scala objects into data structures that are used by the Choco solver.

In Section 6.5 we give an overview of how we modeled our constraints with the Choco library and describe the search strategy that we use.

After the execution of the solver, a printer object is then used to create output files if a feasible solution was found.

The complete source code can be found in the GitHub repository

`https://github.com/AlinaDev/TCAL`

Since the project contains personal information about the teachers of the Otto-Hahn-Schule, the repository is kept private. The project is also available on the CD accompanying this thesis.

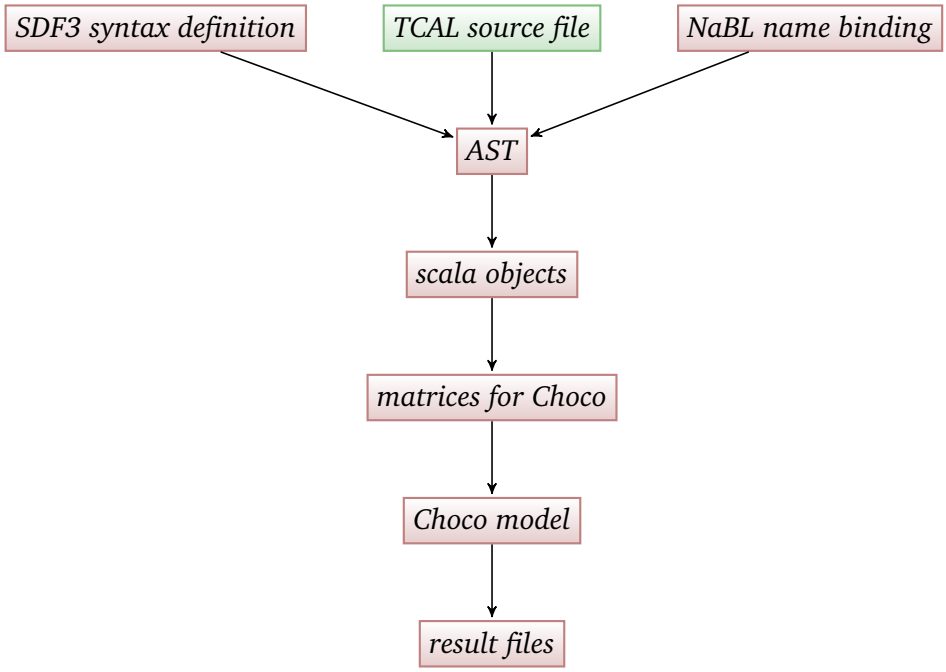


Figure 6.1: Diagram of conversion in TCAL

6.1 Syntax definition

In this section of the thesis the syntax definition of TCAL is explained. The syntax definition formalism SDF3 is used to create TCAL's specification by combining several module declarations. The modules are divided into the overall syntax of a TCAL program, data, constraints, configuration, and a common module that contains the basic language constructs.

A syntax definition of a module in SDF3 can import other modules, as seen in Code Listing 6.1, and use their definitions. This enables separation of concerns and the reuse of modules [11]. The parser uses the symbols defined in *context-free start-symbols* as start symbols and they are the root node of the resulting AST.

The *Common* module contains lexical syntax definitions for low level language elements. These include definitions for identifiers, numbers, comments, layout and whitespace.

We use context-free syntax to define the rest of TCAL. Compared to lexical definitions, context-free syntax definitions are higher-level and can include layout.

A program's grammar is mostly made up of productions, both for lexical as well as context-free syntax. A productive rule has one of the following forms [11]:

<Sort> = *<Symbol>**
<Sort>.<Constructor> = *<Symbol>**

The Symbols on the right of the = define the sort on the left-hand side of the production. The sort can be followed by a *dot* and a constructor name. A symbol's alternatives are comprised of all productions that define its sort [11].

In Code Listing 6.1 the production for a program instance can be seen. An instance has to be introduced by the *instance* keyword and an identification. One or more *Data* elements follow, that have been imported with the Data module. The + represents here, just as in regular expressions, one or more occurrences. The constraints section is mandatory. A configuration section is optional, which is indicated by the *?*.

```
1  module TCAL
2
3  imports
4
5      Common
6      Data
7      Constraints
8      Configurations
9
10 context-free start-symbols
11
12     Start
13
14 context-free syntax
15
16     Start.Instance = <
17         instance <ID>
18             <{Data "\n\n"}+>
19
20             <Constraints>
21
22             <Configurations?>
23     >
```

Code Listing 6.1: TCAL module syntax definition

Within the Data module, productions for the different data elements of TCAL are defined as constructors of the *Data* sort. In Code Listing 6.2 the productions for teachers and classes are shown. The *teacher* or *class* keywords must be followed by an ID for referencing an instance of this object. The body of a data object is comprised of several attributes of the form $x : y$, where the left-hand side x is the name of the attribute and the y on the right of the : separator is a low level language element, like strings or numbers, or the attribute is defined by references to other objects (lines 9, 11, 21, 23 and 24).

Line 9 shows the production that is used to declare the subjects a teacher can teach. In line 26 the used *SubjectRef* sort is defined. References are defined as IDs and are bound to the definition site of the element with the same ID with name binding rules (further explained in Section 6.2).

```
1  context-free syntax
2    Data.Teacher = <
3      teacher <ID> {
4          <{TeacherAttr "\n"}+>
5      }
6    >
7
8    TeacherAttr.Name = <name : <STRING>>
9    TeacherAttr.Subjects = <subjects : <{SubjectRef ", "}+>>
10   TeacherAttr.Hours = <hours : <NUM>>
11   TeacherAttr.AttrType = <type : <Type>>
12   TeacherAttr.Extra = <extracurricular : <NUM> <STRING?>>
13   TeacherAttr.Balance = <balance : <NUM>>
14
15   Data.Class = <
16     class <ID> {
17         <{ClassAttr "\n"}+>
18     }
19   >
20
```

```
21    ClassAttr.Board = <hourboard : <{BoardRef ", "}+>>
22    ClassAttr.Grade = <grade : <INT>>
23    ClassAttr.AttrType = <type : <TypeRef>>
24    ClassAttr.AttrTeacher = <classteacher : <TeacherRef>>
25
26    SubjectRef.SubjectRef = ID
```

Code Listing 6.2: Syntax definition of teachers classes and teacher references

In Code Listing 6.3 the production rules for TCAL's overtime constraint are defined. The *overtime* keyword must be followed by a range of integers and can then be followed by a penalty specification and an option. As described before, the ? indicates that an element is optional.

```
1  context-free syntax
2
3    Constraints.Overtime = <overtime <Range> <Penalty?> <Option?>>
4
5    Penalty.Penalty = [=> [INT]]
6    Range.Range = <[<INT>,<INT>]>
7    Option.Soft = <soft>
```

Code Listing 6.3: Syntax definition of constraints

6.2 Name binding

We reference subjects in the definition of a teacher instance, we reference classes when combining them to a coupling. Correct references from definition to use sites of names are essential to identify the correct program element that is referenced and avoid name conflicts.

Name binding and scoping rules in Spoofax are specified with the DSL NaBL. It is a declarative meta-language where namespaces, definitions, references, scopes, and imports are used to specify a language's name bindings. A compiler then generates a resolution strategy representing the specified name binding [25].

Namespaces can be seen as collections of names that identify elements in a program. The same name in two different namespaces will not cause name conflicts. Even within one namespace name conflicts are avoided, as long as the visibility of the definition sites do not overlap at use site. This is achieved with scoping rules [25].

The term patterns in lines 5, 9 and 12 of Code Listing 6.4 correspond to parts of the program's abstract syntax tree. They are made up of constructors from the syntax definition of TCAL (Section 6.1), variables *(id)* and wildcards *(_)*. Lines 6, 7, 10 and 13 show name binding declarations for the language element corresponding to the term pattern [24].

At the beginning of Code Listing 6.4 a namespace for every data type is introduced. The binding rule in line 6 declares the definition site of an instance, where *id* represents its name.

The scoping rule in line 7 restricts the visibility of all data types within one instance from being seen outside of the instance. Therefore several instances of assignment programs can be in one project without causing name conflicts. As a result one program can be duplicated and used as the basis for next years teacher assignment with only the necessary updates and a new instance name.

Lines 10 and 13 show the difference between a definition site and a use site. While line 10 indicates the definition site and the teacher's *id* is being bound to the corresponding program element, line 13 shows a reference to the definition site of a teacher.

```
namespaces
    Instance Data Teacher Subject Class Coupling Hourboard Type

binding rules
    Instance(id, _, _, _) :
        defines Instance id
        scopes Teacher, Subject, Class, Coupling, Type, Hourboard,
           Preassignment

    Teacher(id, _) :
        defines Teacher id

    TeacherRef(id) :
        refers to Teacher id
```

Code Listing 6.4: Name binding and scoping rules in NaBL

6.3 Editor Services

As discussed in Section 2.1.3, Spoofax parses and analyzes the source files on the fly and default editor services are provided according to the specified syntax and name binding. To support the users of TCAL, we improved the generated outline view and syntax highlighting.

Outline View

The data constructs in TCAL can be defined in the order that makes most sense to the user. Even though having all hour boards in one part of the program eases maintaining them, it can be practical to define hour boards close to a class definition if the hour board is only ever assigned to this class.

However, this individual ordering can complicate finding a specific element. Spoofax automatically creates an outline view of the program with the elements' IDs as labels. The identifiers we used for the exemplary school are often very similar to each other and can even include duplicates if the type of data element is not the same. By adding icons to those labels depending on the data construct the ID refers to, an element can easily be found in the outline view (Figure 6.2).

Syntax Highlighting

The generated syntax highlighting did not highlight referenced program elements. We added highlights for all IDs (lilac), strings (blue) and numbers (green).

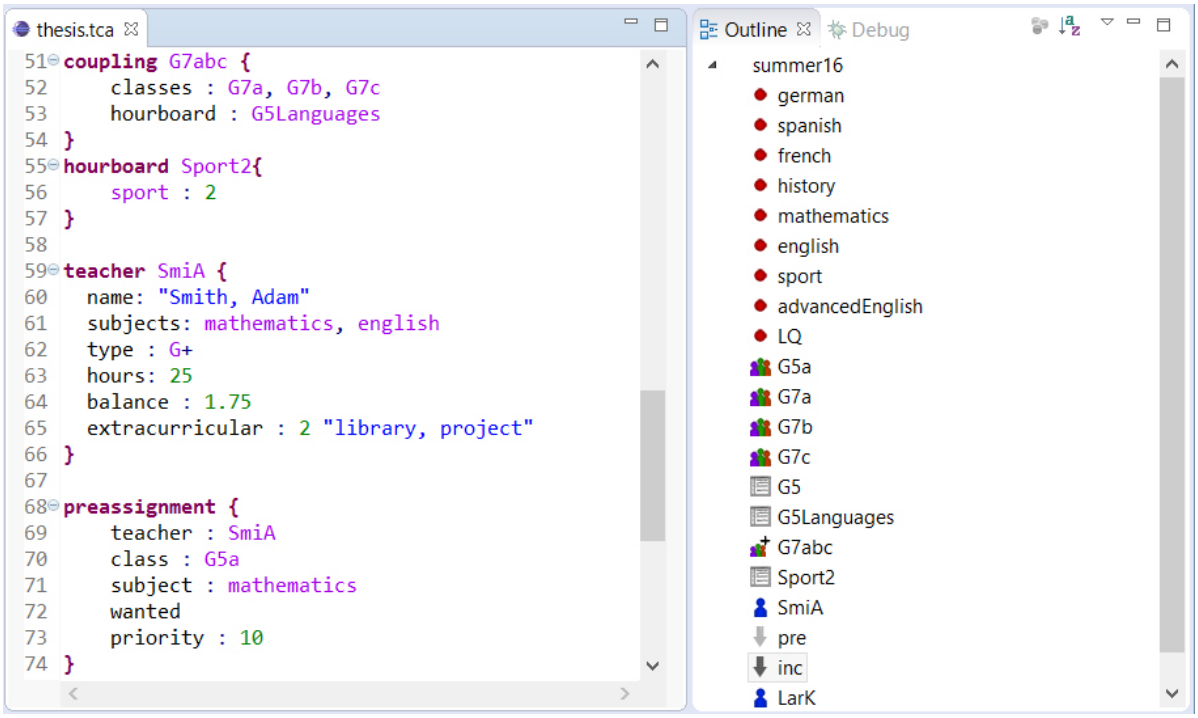


Figure 6.2: Syntax highlighting and customized outline view

6.4 Compiler

As described in the requirements for TCAL, extending the language and using other tools for solving the underlying algorithmic problem should be as easy as possible. Spoofax offers the possibility to dynamically load a Java strategy in form of a Java archive instead of writing code transformations and generations with the Stratego language within the Spoofax project. Taking advantage of this feature, and the compatibility of Scala with Java, we wrote TCAL's strategy in Scala. Within the Scala strategy the AST that is created from a TCAL source file is converted to Scala objects. Only then is the relevant data used to model the algorithmic problem of finding the best assignment with the help of the Choco solver.

This compilation in several phases is useful to offer optimal structures for the user and for the solver. While a separation of the data into hour boards, classes and couplings simplifies maintenance for the user and gives the program a more expressive structure, the solver only needs teachers and lesson with their individual attributes and constraints. The intermediate data representation as Scala objects is useful if the solver tool is exchanged, as they are easily converted into the required format for any tool.

6.4.1 AST to Scala Objects

After parsing, desugaring, and analyzing the contents of a TCAL source file, an AST of the program is used as the input for the Scala strategy. The AST is first converted into a tree of StrategoTerms (Code Listing 6.5), namely

StrategoAppl, representing all language elements for which a constructor is defined in the language's context-free syntax

StrategoString, containing an ID, a string or a number

StrategoList, containing a list of StrategoTerms

```
StrategoAppl(Instance,WrappedArray(
  StrategoString(summer16), StrategoList(ArraySeq(
    StrategoAppl(Teacher,WrappedArray(
      StrategoString(SmiA), StrategoList(ArraySeq(
        StrategoAppl(Name,WrappedArray(StrategoString("Smith, Adam"))),
        StrategoAppl(Subjects,WrappedArray(StrategoList(ArraySeq(
          StrategoAppl(SubjectRef,WrappedArray(StrategoString(mathematics
            ))),
          StrategoAppl(SubjectRef,WrappedArray(StrategoString(english))))
            ))),
        StrategoAppl(AttrType,WrappedArray(
          StrategoAppl(TypeRefPlus,WrappedArray(
            StrategoAppl(TypeRef,WrappedArray(StrategoString(G)))))))),
        StrategoAppl(Hours,WrappedArray(StrategoString(25))),
        StrategoAppl(Balance,WrappedArray(StrategoString(1.75))),
        StrategoAppl(Extra,WrappedArray(StrategoString(2),
          StrategoAppl(Some,WrappedArray(StrategoString("library")))))))
            ),
        [...]
    )
)
```

Code Listing 6.5: AST of StrategoTerms (syntax constructors in bold)

Within the Scala strategy, pattern matching is used to traverse the AST of StrategoTerms and convert them into Scala objects. Code Listing 6.6 shows the beginning of this process. The *Instance* is always the root of a TCAL AST. It is followed by the instance ID (*StrategoString(instance_id)*), a list of data definitions (*StrategoList(dataStrat)*), a list of constraints (*StrategoList(constraintsStrat)*) and an option for configurations (*configuration*) in line 2. The data, constraint and configuration elements are then traversed in separate functions to create the respective Scala objects.

```
1  input match {
2      case StrategoAppl("Instance", StrategoString(instance_id),
           StrategoList(dataStrat), StrategoAppl("Constraints", StrategoList(
           constraintsStrat)), configuration) => {
3        instance = instance_id
4        configuration match {
5          case StrategoAppl("Some", StrategoAppl("Configurations",
               StrategoList(configs))) => convertConfigurations(configs)
6          case StrategoAppl("None") =>
7          case _ => throw new Exception("Unexpected configuration format")
8        }
9        convertData(dataStrat)
10       convertConstraints(constraintsStrat)
11       print("constraints: " + constraints + "\n")
12     }
13     case _ => throw new Exception("Unexpected AST")
14   }
```

Code Listing 6.6: Traversing Stratego AST

Chain of References

As mentioned before, data elements in TCAL often reference each other. Trying to convert a teacher definition into a Scala object without having converted the teacher's subjects beforehand would result

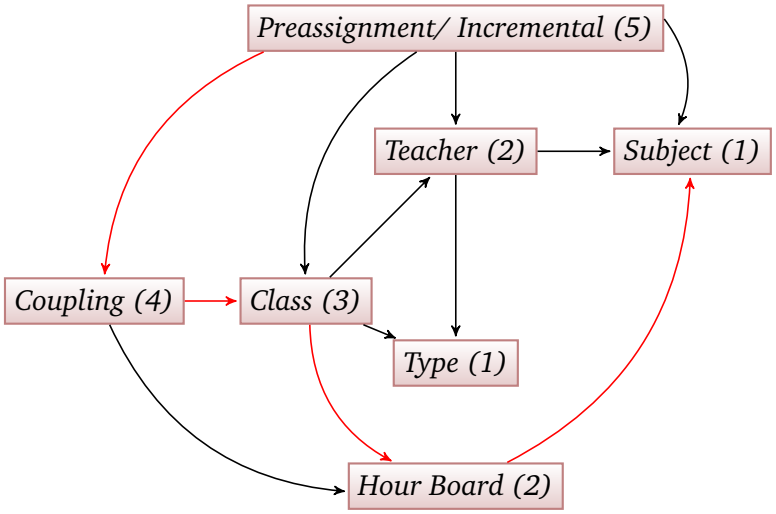


Figure 6.3: Illustration of what object references another and resulting order of conversion

in problems. One alternative of handling this would have been to force the user to define the data in a specific order, which would have been an unnecessary complication in our eyes. Instead we traverse the data subtree of the AST in five rounds.

In Figure 6.3 the chain of references is illustrated and the number behind each element represents the round in which this element is converted. The red path is a longest path within the graph from an element that is not referenced by any other (last one to be converted) to an element that does not reference any other (first one to be converted). This path has five edges, therefore we need at least five rounds.

Lessons

In TCAL's data structure lessons can be inferred by the combination of a class or coupling object and their referenced hour boards. By this the maintenance of a grade's hour board is simplified and copying the same lessons to every class is avoided.

However, to model the problem for the solver, this structure does not make sense and would lead to unnecessary complication. During the third round of the conversion, classes and couplings are converted. For every entry in their referenced hour boards *Lesson* objects are created, referencing the taught subject, its class(es) and the amount of hours the lesson is taught per week. Couplings are from now on represented by a set of classes.

A lesson also has an option for a link to other lessons. This option is used when the lesson's subject extends another subject. If both subjects are taught in the same class they are both taught by the same teacher. Coordination lessons are also linked to other lessons (Section 4.2) to constrain the teachers allocated to the linked lesson to be assigned to the coordination lesson.

Incremental Preassignments

Assignments from a previous execution of a program in TCAL can be used as mandatory assignments in the next execution. As described in Section 4.2, one or more references have to be specified in an incremental preassignment object to determine which assignments should be retained. During the fifth round of the conversion to Scala objects, all lessons are filtered out that are affected by this incremental preassignment. For each one of those lessons, a mandatory preassignment is created with the teacher from the previous assignment.

6.4.2 Scala Objects to Matrices

After the AST of Stratego terms is converted into Scala objects, they can be used as the data basis for a tool to solve the constraint optimization problem of the actual assignment. For this task we use the Choco solver, described in more detail in Section 2.2.4. To build the model with the Choco library we convert the Scala objects into matrices containing all relevant information for the solver. This way information can be quickly looked up.

6.5 Modeling and Solving the CSOP

We use the java Choco library to solve the constraint optimization problem of assigning teachers to classes and lessons. Choco is separated into a modeling part and the solver. In this section we will describe how a TCAL program's constraints are modeled with the Choco library and the search strategy is described that is used by the solver to calculate the best teacher allocation.

6.5.1 Modeling Constraints with Choco

The first instruction to model a problem with Choco is to create a *Solver* object (Code Listing 6.7 line 1). The *Solver* manages the variables, constraints and it guides the search loop.

In constraint programming, variables are the unknown that have to be assigned a value, obeying all constraints, to get a feasible solution. In lines 3 and 4 a matrix and an array of variables are declared. The *as* matrix contains the solution's assignment of teachers to lessons. The rows of *as* correspond to all teachers and the columns to all lessons. For every variable a domain of possible values must be defined. The variables in *as* are *BoolVar*, a special *IntVar* with a fixed domain of 0 and 1. In a feasible solution the lesson with index i is assigned to the teacher with index j if the value 1 is assigned to the vaiable *as[i][j]*, otherwise 0 is assigned.

For a normal *IntVar* the variable's domain has to be defined. In line 4 a variable array for the overtime penalties is created. It has an entry for every teacher and the domain of each entry is between 0 and 500.

In line 7 we iterate over all mandatory preassignments. A preassignment can be wanted or unwanted, which is specified by *assigns[a][2]* being 1 or 0 respectively. We then post a constraint to the solver, that the teacher with index *assigns[a][0]* is assigned to the lesson with index *assigns[a][1]* by constraining the entry *as[assigns[a][0]][assigns[a][1]]* of the assignment matrix to be equal to *assigns[a][2]*. This is an example of a unary hard constraint. Every created constraint only involves one variable (*as[assigns[a][0]][assigns[a][1]]*) and if the constraint is not satisfied, a solution is not feasible.

```
1 Solver solver = new Solver("Assigner");
2
3 BoolVar[][] as = VariableFactory.boolMatrix("assignment", num_teachers,
     num_lessons, solver);
4 IntVar[] pen_overtime = VariableFactory.boundedArray("penalty overtime",
     num_teachers, 0, 500, solver);
5 IntVar score_overtime = VariableFactory.bounded("score overtime", 0,
     100000, solver);
6
7 for(int a = 0; a < assigns.length; a++){
8     solver.post(ICF.arithm(as[assigns[a][0]][assigns[a][1]], "=", assigns
          [a][2]));
9 }
```

Code Listing 6.7: Variable creation and mandatory preassignment constraints

The overtime constraint showcases a variety of constraints and modeling techniques. In Code Listing 6.8 line 3 a variable is introduced that is constrained to contain the amount of hours a teacher is scheduled to teach by posting the higher-order constraint in line 5. *as[t]* is an array of 0's and 1's and *hoursLessons* is an array where the entry *hourslessons[i]* represents the amount of hours the lesson with index i is taught in a week. The scalar product, also called the dot product, evaluates to the scheduled hours of the teacher.

The range that is specified by the user for the allowed overtime is saved in the array *overtime*. A teacher is allowed to teach at most *overtime[0]* less and *overtime[1]* more hours per week. The constraints in lines 6 and 7 enforce these hard constraints.

Additionally, soft constraints are required to favor solutions that involve as little overtime as possible. To represent soft constraints we introduce penalty arrays. Every entry of the penalty array *pen_overtime* represents the overtime of one teacher. We use *variable views* in line 8 to define the entry as a function of the absolute difference between the scheduled and desired hours, multiplied by the penalty saved in *overtime[2]*.

score_overtime is then constrained to be the sum of all entries in *pen_overtime*.

The objective function of our model is to minimize the sum of all scores.

```
1  for(int t = 0; t < num_teachers; t++) {
2      //every teacher works as many hours as desired, at most overtime[0]
           less or overtime[1] more
3      IntVar scheduledHours = VariableFactory.bounded("scheduled hours", 0,
           500, solver);
4      int desiredHours = teachers[t][num_subjects+1];
5      solver.post(IntConstraintFactory.scalar(as[t], hoursLessons,
           scheduledHours));
6      solver.post(IntConstraintFactory.arithm(scheduledHours, ">",
           desiredHours + overtime[0] - 1));
7      solver.post(IntConstraintFactory.arithm(scheduledHours, "<",
           desiredHours + overtime[1] + 1));
8      pen_overtime[t] = VariableFactory.scale(VariableFactory.abs(
           VariableFactory.offset(scheduledHours, -desiredHours)), overtime
           [2]);
9  }
10 solver.post(IntConstraintFactory.sum(pen_overtime, score_overtime));
```

Code Listing 6.8: Model of the overtime constraint in Choco

6.5.2 Search Strategy

As described in Section 2.2.4, the Choco solver uses search strategies to find a solution or determine that there is no feasible solution. During the search a binary search tree is constructed, adding a node for every decision that is made. A decision is usually comprised of the assignment of a value to a variable. Which variable is assigned next and what value is selected from its domain depends on the specified search strategy. The decision is then propagated. If a decision causes a failure, backtracking is used to undo the decision according to the binary search tree and a different decision is made. By doing so the search space is explored with a depth first search. If no more free variables are available, either a solution has been found or no feasible solution is possible [30].

The Choco solver offers a number of built-in search strategies. Using the *domOverWDeg* strategy provided us with the best results.

domOverWDeg

```
solver.set(IntStrategyFactory.domOverWDeg(ArrayUtils.flatten(as), System.
    currentTimeMillis()));
```

This search strategy is based on the adaptive variable ordering heuristic introduced by Boussemart et al. [4]. By maintaining information about previous states of the search, the heuristic can guide the search toward hard parts of the problem. To do so, every constraint has a weight associated to it. This weight is initialized to 1 and is increased every time the constraint is the cause of a backtrack. Every variable is then attributed a weighted degree (*wdeg*), that can be constructed by the sum of the weights of all constraints that involve the variable and at least one other uninstantiated variable. The weighted degree is combined with the variable's current domain size (*dom*) into $\frac{dom}{wdeg}$. The variable with the smallest ratio is assigned next. At the beginning of a search, *wdeg* represents the number of constraints the variable is involved in. The heuristic is therefore initially a combination of the most-constrained-variable and the most-constraining-variable heuristic (see Section 2.2), as it chooses the variable with the fewest possible values and that is at the same time involved in the most constraints. However, as the heuristic collects information, variables that are involved in constraints that are hard to satisfy are assigned with a higher probability.

With extensive experiments Boussemart et al. showed in 2004 that the approach was the "most efficient current one with respect to significant and large classes of academic, random and real-world instances" [4].

7 Evaluation

The goal of this thesis was to create a way to support schools during teacher allocation. There is a variety of software available to compute the timetable of a school but none to automatically assign teachers to classes and lessons. We therefore decided to design a DSL and implemented it in the course of this thesis.

To evaluate the result of our project, we will at first describe the process of the language development. By explaining design decisions, the incremental addition of new language constructs and the reasons for each, we will justify the language's syntax. We will evaluate our solver model and compare our language to the requirements that we set ourselves in the concept. Finally we will evaluate the resolution performance and the resulting assignment.

7.1 Requirements Analysis, Conceptualization and Basic Language Development

To gather all relevant constraints of the problem at hand, we talked to domain experts from a local school. They explained to us all regulations and reasonings that they have to consider when assigning teachers to classes. With this information we created an initial concept. It contained a list of all constraints, divided into hard and soft constraints, and an overview of the data structure we thought was needed to implement them. The concept was then improved during the course of this project and evolved into the concept that can be found in Chapter 5.

One question that we had to decide was how free the user should be to define constraints. We decided to offer a limited amount of built-in constraints where the user could then specify if the constraint has to be enforced. The user can also customize the constraint's parameters if necessary or desired. This enables users with little programming experience to use the language.

We started out by defining the most basic entities necessary for a teacher assignment; teachers, classes, subjects and hour boards. Hard constraints for the number of teachers per lesson, the constraint that a teacher must be qualified to teach a subject and minimal and maximal overtime were added. To minimize the amount of overtime, penalties for the resulting overtime were introduced. The solver was set to minimize the score of a solution, the sum of all penalties.

For the syntax of the language we used the domain's terminology and went without using common programming keywords. Attributes are defined in a way that is familiar from filling out a form in real life.

To test our language with realistic data we then received an excel sheet with the school's teacher data and hour boards. This data was unfortunately not completely clear without further knowledge of the teachers and specific regulations. This strengthened our desire to offer an unambiguous data representation with our language.

A pastor was for instance listed to teach religion in Hauptschule and Realschule. The number of hours that could be taught was however significantly less than the amount of hours he was supposed to teach. No solution could be found, because the pastor could not be assigned an adequate amount of hours.

Furthermore, the distribution of lessons onto classes was not apparent from the hour boards. For lessons like religion or languages the classes are often mixed together. This could not be deduced from the data given to us.

With a pdf of the current teacher assignment, most questions could be answered. We could infer the hour boards, added subjects and project groups, and corrected the teacher data.

7.2 Process of Adding New Language Constructs

In this section we will explain the reasoning behind the language constructs that were incrementally added to the language.

Students do not have all lessons within their usual class. For lessons like religion, languages or elective courses students from different classes come together. At first we had planned to create fake classes for these occasions. The disadvantage of this would be, that there would be no reference to the real classes that partake in the lessons. Instead we added the language construct of couplings. A coupling references the classes it unites and the hour boards that are assigned. After finding a solution we can print out the assignment and group it by classes, with couplings' lessons showing up in the sections of every class of the coupling.

As mentioned before, even though teachers are trained to teach one type (HR or G), they are often allowed to teach other types. To facilitate this in our language we added an *all* type. This type is built-in and allows the teacher to be assigned to lessons of all types.

In Germany all classes have one class teacher and they usually have periods that are called home room where problems can be discussed, projects or excursions are planned. We added a built-in subject *homeroom* and the *classteacher* attribute for the *class* definition. All lessons with subject *homeroom* are constrained to be assigned to the class's class teacher.

After viewing the current teacher assignment we realized that there were a number of subjects that differ from the usual subjects that teachers can study for at university (e.g. mathematics, German, biology or history). Therefore these subjects are not referenced when a teacher is defined. To check if a teacher is qualified to teach a lesson, the lesson's subject and the subjects referenced when defining the teacher are compared. As as result an assignment of these subjects was not possible. These subjects include special science courses and courses that practice reading, writing, and arithmetic. When asking the domain experts they explained to us what teachers are qualified to teach each subject. Now we had to find a way to include these subjects with their real name, but also check which teachers are qualified to teach them. We added the possibility to extend subjects. This way a teacher that is allowed to teach the extended subject is also allowed to teach the extending subject. For example, the subject where reading, writing, and arithmetic are practiced extends German, hence all German teachers are allowed to teach it, without having to reference the subject when defining the teacher.

A class teacher often teaches several subjects in his class. This can be demanded by defining preassignments, but we decided to add a hard constraint that at least one lesson must be taught by the class teacher.

Some teachers take on tasks besides teaching. These extracurricular activities include library duty and the tasks of a principal, vice principle or guidance counselor. Decreasing the teacher's desired hours in his definition would result in a wrong data representation. Instead we added the *extracurricular* attribute to the teacher definition with an optional label. The hours specified as extracurricular are subtracted from the teacher's available hours per week when converting the Scala objects to matrices for the Choco solver. The label can be used when printing out the solution. While listing all lessons that are assigned to one teacher, the extracurricular activity can thereby be listed as well with the correct label.

If the solver can not find a solution, the user usually does not get any feedback as to what the reason for it might be. We therefore added the possibility to turn the hard overtime constraints into soft constraints with a very high penalty. This however results in a very bad resolution performance. When the

input size of the problem is of bigger proportion this option can not be used to find an optimal solution. However, instead of not getting any solution and feedback, the user gets a solution, which is far from optimal, but he sees what subject or teacher causes the problem.

A teacher's desired hours, balance or extracurriculars are not always whole numbers. We added the possibility to use numbers with decimal places (floats) in these cases. However, because lessons' hours are always whole numbers, we do not need the partial hours to have correct hard constraints for overtime. The teachers' hours are therefore rounded down for the solver model. The only problem is that the penalty does not contain the partial hours, which slightly distorts the score of a solution. When we tried changing the penalty variable from an integer into a number with decimal places, many constraints could not be used in Choco anymore and we could not create an overall score.

We previously added the built-in type *all* because we realized that teachers may teach other types than they were originally trained to. In most cases teaching in a class of their original type is preferred, which could not be represented by using the *all* type. We therefore introduced the option to make a type a "plus type" by adding a + to a teacher's type. Teachers with "plus type" are allowed to teach other types. To control this procedure we added a customizable type constraint, so that the user can for instance specify that "plus teachers" of type A can teach classes of type B. Furthermore, the user can limit the grade in which the constraint takes effect and he can specify the penalty that should be added if a "plus teacher" teaches another type if desired.

It is obvious that not all teachers can teach exactly the designated amount of hours per week. For this and various other reasons teachers may have overtime from previous years. This overtime can be defined as balance in the teacher definition. If a teacher has already a positive or negative balance, it should be preferred to decrease it and avoided to increase it. Our first intuition was to add the balance to the desired hours and use the new number for the solving process. However, the balance can be rather large for some teachers and it may not be viable to work it off in one school term. This being the case, we decided that the user could specify a second range that limits the maximum deviation from the original desired hours (details in Section 4.3).

If a teacher works many hours in the upper classes (grades 10 to 13) he is credited additional hours, because preparations are more elaborate. Defining a syntax for this constraint was the hardest of all constraints. There are many parameters of the constraint that a user could customize; the teacher's type, affected grades, hours that are credited, threshold value of hours when the additional hours are credited. We decided to limit the customization to the credited hours and the threshold.When similar constraints are added in the future, it could be sensible to adjust this constraint. Furthermore we did not implement partial credit of hours for part time teachers, because partial hours are not supported in our Choco model.

7.3 Modelling the Problem

The focus of this project was the development of a language. Unfortunately we did not have time to optimize its resolution process completely.

Modeling simple problems with Choco is very intuitive, but it was hard to find good documentation or examples about modeling with many soft constraints and complex constraints with many intermediate variables.

As described before, we could not add partial hours for the teacher. Using number variables with decimal places for the overtime penalty would have inhibited us from using necessary constraints.

The range for the hard overtime constraint must be small to find a good solution in a reasonable time. Even though the quality of the solutions will also increase over time with a bigger range, the resolution performance suffers from the additional possibilities.

7.4 Results

Fulfillment of Requirements

Our goal was to design a language that could express all constraints that were given to us by the exemplary school and solve the modeled problem. Moreover, the language should fulfill the following requirements as stated in Section 3.2:

- **Readable and understandable without programming experience**
  We used domain terminology and intuitive syntax to design TCAL.
- **Unambiguous and expressive data representation**
  The prior representation of data with Excel spreadsheets was not clear without further insight, because classes were not represented individually, not all hour boards could be deduced. It was not clear which teachers are allowed to teach special subjects. In TCAL every class is defined individually. Hour boards can be assigned to several classes or couplings, which helps with maintaining consistency and gives a clear representation of each class's lessons. Because special subjects must extend another subject, it is always clear which teachers are allowed to teach it.
- **Minimize the amount of coding effort**
  Necessary constraints for our exemplary school are built-in and only have to be customized. With the possibility of assigning hour boards to several classes, the amount of boiler-plate code is further decreased. Spoofax's generated editor service of code completion also takes a lot of the coding work out of the user's hands if desired.
- **Purely declarative modeling**
  All algorithmic details, constraint implementations and data conversions are hidden from the user. Instead of defining the necessary steps to solve the problem, the user only has to define the data elements, customize constraints or specify if a constraint must be obeyed.
- **Language easily extensible**
  Thanks to Spoofax and the segmentation of the compiler into two phases, extending the language can be done with little effort. When modifying the language's grammar and after updating the name binding rules, Spoofax will keep the rest of the language definition consistent. Adding elements to the Scala strategy with its clear structure is also straightforward.
- **Efficient implementation**
  The Choco solver that we chose to solve the teacher allocation problem is one of the fastest on the market. However, the resolution performance is not completely satisfactory. The solver sporadically does not find any solution. A more specialized model or algorithm could improve TCAL.
- **Problem solving tool effortlessly replaceable**
  Because the AST that is created by Spoofax is converted into descriptive Scala objects, they can easily be converted into any data format suitable for other problem solving tools. Testing different solvers could improve the resolution performance.

Resolution Performance

We tested our language with the data given to us by the Otto-Hahn-Schule. The problem is comprised of 114 teachers and 888 lessons. With the constraints listed in Figure 7.1, 214.270 variables and 94.004 constraints are created from the Choco model.

	exemplary school
# teachers	114
# lessons	888
specified constraints	overtime [-3,2] add balance [-5, 3] G can teach HR =>2 HR can teach G [5,10] =>3 add classteacher lesson add 1 hour when 8 hours in upper
# variables	214.270
# constraints	94.004
first solution found	ca. 30 sec
search completed	-
best score found	265

Figure 7.1: Solution statistics for exemplary data

The solver was run on a computer of the following properties: Windows 8.1, Intel (R) Core^{TM} i7 2.50Ghz CPU with 16GB RAM.

The first feasible solution is usually found after around 30 seconds, as seen in Figure 7.2. Each graph in this figure represents one execution of the program, where points stand for a solution that was found. However, after the first series of solutions is found, the solver does not find another solution for long stretches of time. Sporadically the solver does not find any solution within the time limit.

Eight hours was the longest period of time that we let the solver run. The optimal solution was not found yet.

Choco does offer a lot of ways to customize the solver, but these possibilities are not well documented. For a lot of more complex search strategies the parameters are not explained, therefore we had to specify them as best as we could. We did test all build-in search strategies, but none could improve the result presented here.

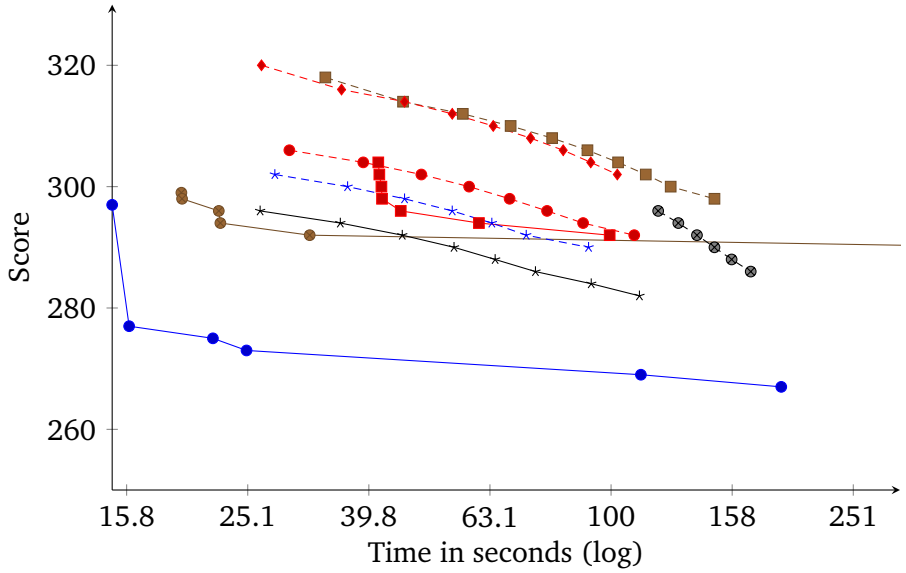


Figure 7.2: Score depending on solution time (every graph represents one resolution round)

8 Related Work

In literature the assignment of teachers to courses is mostly found in combination with the timetabling problem. The creation of timetables is such an omnipresent problem that the research about it is countless. To share latest results and exchange ideas, the first international conference on the Practice and Theory of Automated Timetabling (PATAT) [1] was organized in 1995 for researchers and practitioners to meet. The conference is held every other year ever since and one topic that has been revisited from the beginning is the creation of a standard timetabling language or format.

Because the teacher allocation problem, however, varies greatly around the world, we could not find any research about the development of a standard language or format. The research in this area is limited to algorithms solving problems similar to the one covered in this thesis.

In this chapter we will give an overview of the research related to the thesis's topic. We will present some of the most influential language designs and implementations for the timetabling problem. Subsequently we will discuss algorithms that were introduced to solve teacher allocation problems.

8.1 Languages, Data Formats and Frameworks

For specific problems researchers often do not focus on a general model, but instead focus mainly on solving the problem efficiently, resulting in very specialized and optimized solutions. When using very specialized models, small changes in the specification of a timetabling problem can result in the need for radical changes in the data structure and the used algorithm. Furthermore, comparing results and benchmarking becomes almost impossible. On the other hand, a standardized format for such a diverse problem results in a more abstract and less expressive language. If the level of abstraction is too high, the use of solution methods specialized for particular versions of timetabling may be prevented [23].

The motivation for a standard language or format is better comparability and transferability. An archive of freely available test data could be assembled and used for comparable testing of new algorithms and strategies.

Burke, Kingston and Pepper specified a list of requirements for a standard data format [6]. The format must be general so that all possible instances can be described in the language. It must be possible to formulate instances completely, including expressing all data constructs, constraints, optimization criterion and the proposed solutions. Furthermore, it must be accessible so that it is easy to translate to and from.

We have found that the only format that is still being used is an XML format introduced by the Benchmarking Project for (High) School Timetabling (see below).

TTL1

For their computer program for high school timetabling, Cooper and Kingston introduced the language TTL1 [8] in 1993 to specify instances of the timetabling problem. Unlike many prior proposed languages, the assignment of teachers to lessons does not have to be fixed before computing the timetable. However, only a basic set of constraints is available from the language specification and the expression of preferences is not possible.

RAPS

RAPS is a "Rule-Based Language for Specifying Resource Allocation and Time-Tabling Problems" [36] that was introduced in 1994. An instance of a problem can be specified by defining its resources, activities, allocation rules and constraints. The user can also specify control and backtracking strategies. The compiled program is then run by the "expert system for resource allocation" (ESRA) engine, a rule based expert system.

TTLIB

Burke, Kingston and Pepper developed 1998 [6] a data format for timetabling instances. TTLIB is intended for comparison of results, exchanging data and using shared data for algorithm benchmarks and evaluation. It is not intended to be a language from which programs are written. To allow every possible constraint to be expressed, the format is based on set theory and logic to formulate all constraints as functions. Representing complex constraints in this format is difficult, so that the creation of library files of complex constraints is suggested.

STTL

The language, introduced by Kingston in 1999, is object-oriented and functional [19][20]. It was intended for specifying and evaluating timetabling problems, their instances and solutions. Time, resource and meeting objects define the data of a problem and functions are used to express constraints. With some specialized features, STTL could be used to model any complex assignment problem. The author also provided an interpreter for the language. However, it is not easy to convert existing data to STTL [27].

UniLang

Intended to be an input language for any timetabling system, UniLang was introduced in 2001 [32]. Its representation of a timetable instance was meant to be understandable for both computer specialist and school administrators. Unlike the majority of proposed languages or data formats, UniLang kept the different versions of the timetabling problem (e.g. high school, university, examination) in mind during the design and was intended to be suitable as an input language for any timetabling system.

Standard Framework and GTL

The standardized framework [14] introduced in 2002 can be used to describe many different kinds of timetabling problems, not just school timetabling. It offers standardized input and output formats. With the framework, the general timetabling language GTL is introduced with syntax similar to Java. While the problem is described using GTL, the data of a specific instance must be converted into an XML input file. Standardized timetabling algorithms are part of the framework and the result can be exported to XML, HTML or text format.

TTML

The Timetabling Markup Language (TTML) is a XML data format based on MathML introduced in 2005 [27]. The timetabling problem is specified using set theory, where functions on those sets represent constraints. The output and test results to a timetabling problem can also be modeled with TTML. A multipurpose TTML processor, including parser, problem solver and a solution interpreter was a future goal.

Architecture for Workflow Scheduling

With the 2005 introduced architecture, workflows can be modeled and scheduled, obeying resource allocation constraints as well as temporal and causality constraints [2]. Resources are assigned to tasks and the order of task execution is scheduled. The application area is very general, including all business processes than can be modeled as workflows. The Workflow specification language (WSL) is introduced to specify the resources of a problem and the resource allocation constraints on them. The architecture also contains a scheduler module with a constraint solver to find a solution to the resource allocation problem, using the constraint programming language *OZ* for the implementation.

KTS

KTS is a web-based software system, introduced in 2007 by Kingston, the author of TTL [8] and STTL [20], that can be used to solve high school timetabling problems [21]. It includes a web server, user

interface and a solver. The KTS data model is only used within the system. The software system is still online.

An Extensible Modeling Framework

In 2007 Ranson and Ahmadi suggested a different approach to standardize the modeling of timetabling problems. Instead of creating a new language they introduced a standard modeling framework [31]. The framework can be extended and incorporates features of object-oriented programming and UML.

Benchmarking Project for (High) School Timetabling

In 2008 a group of researchers founded the *Benchmarking Project for (High) School Timetabling*. In the following years they established a standard XML format and provided an archive of datasets to enable freely available test data and comparable benchmarks.

The XML format was first introduced in 2008. The version described in 2011 [29] is the format used for the benchmarking project today. Authors that introduced data formats and frameworks in previous years collaborated on this project, namely Ahmadi and Ranson (extensible modelling framework [31]), and Kingston (TTL [8], STTL [20] and KTS [21]).

An instance is described by *time, resource, event*, and *constraint* elements in the XML file. All restrictions and requirements on the timetable are limited to the constraint part of the specification. This way the data declarations are sufficient to represent data for various countries, while the constraints can be extended if need be without changing the structure of the data representation.

The data format has only a declarative purpose so that instances in this format can be used as input for timetabling systems and search strategies.

In 2010 the first archive HSTT2010 was introduced with 15 instances from 7 different countries. In 2016 the latest version of the archive XHSTT-2014 contains around 50 datasets from several countries and varying significantly in size [28].

The XML format also models solutions. For benchmarking, the solutions can then be evaluated with Kingston's HSEval High School Timetable Evaluator [22]. The evaluator verifies the compliance with the XML format, provides the infeasibility and objective values that represent how many hard constraints are violated and how well soft constraints were satisfied, and finally it compares solutions if multiple solutions were included.

8.2 Algorithms

An Operations Research Approach

Tillett was the first to describe an algorithm for the assignment of teachers to courses in a secondary school [38] in 1975. His starting position was that teachers had to be assigned to courses and the courses had to be assigned to rooms and time slots before distributing the students to the courses, according to their requests. The algorithm was based on the operations research technique of linear programming. In four of seven test cases the resulting solution was superior to existing schedules in regards to their preference and effectiveness ratings. However, the resolution time with the integer programming algorithm was too great for departments of bigger size.

A Linear Programming Solution

A program was introduced in 1976 for simultaneously solving all subparts of the timetabling problem at a university, namely the determination of which courses should be offered and how many sections in each, the allocation of faculty to courses, the assignment of the courses and sections into timeslots and rooms, and lastly the distribution of students to courses and sections [5]. After an initial processing a listing of possible assignments was produced and only after communication with the administration, was the problem solved completely.

A Special Case of the Fixed Charge Transportation Problem

The problem of assigning classes to professors at a university is discussed with the objective of minimizing the average number of distinct subjects assigned to each professor. The teacher assignment problem is formulated as a fixed charge transportation problem [16]. In comparison to the solutions of a general mixed integer program solver, the solutions of this specialized algorithm are superior.

Genetic Algorithm

Wang proposed 2002 a genetic algorithm to find a solution to the teacher assignment problem faster and with a higher satisfaction rate of the teacher's preferences [41]. To do so, the offered courses, the teacher's qualification and preferences, the minimum required teaching hours and the limit of overtime hours are considered. The goal was to find a fair distribution of overtime and to fulfill the teachers' preferences as satisfactory as possible. To find a solution, a genetic algorithm, which is based on the biological principles of selection, reproduction, and mutation, is used. Genetic algorithms are global search algorithms that process combinations of parameters rather than single parameters. After a genetic representation of the problem is found, genetic operators are used to find new combinations of parameters. According to a fitness function the combination can be evaluated.

Tabu Search

To assign teachers to subjects and groups in a secondary school in Spain a constructive procedure is firstly used to find an initial solution and with a tabu search algorithm the solution is improved [3]. The teacher assignment is used as a first phase to the timetabling problem, and the assignment is evaluated by the quality of the resulting timetable.

Hybrid Algorithm

A hybrid algorithm, combining an integer programming approach, a greedy heuristic and a modified simulated annealing algorithm, is used to solve the teacher assignment problem and the course scheduling problem simultaneously [15]. The results indicate that the algorithm can handle large data sets better than previous proposals. The problem is firstly specified as a mathematical programming model and coded in C++ for testing.

9 Future Work

One of the requirements for our language was that it could be easily extended to fit different constraints and regulations that may be relevant at other schools. Furthermore the implementation was designed to effortlessly exchange the tool that is used to solve the underlying allocation problem. Following we will describe possible improvements and additions to our language.

Constraints and Model

Partial hours are currently not reflected in the teachers' overtime penalties. Finding a way to represent them would be favorable in the future.

In our model all lessons are assigned to exactly one teacher. The coordination hour is the only exception. This could easily be adjusted in the future. A lesson could be assigned to more than one teacher, for example for a project group or if an assistant teacher has to be overseen by another teacher. Even though there was no use case for that at our exemplary school, it is nevertheless already built in to the constraint model. Therefore only the language syntax would have to be adjusted to accommodate specifying the number of needed teachers.

Different teachers vary in their style of teaching. Teachers could be grouped by their style of teaching and an additional constraint could be to avoid having too many teachers of the same kind teach the same class.

When there are not enough teachers to cover all lessons, the school can hire a new teacher. With the current language, the user could create a new teacher and try out different subjects to see what combination would result in the best assignment. A built-in mode that would compute the best combination of subjects a new teacher would need could be a good addition to the language.

Assignment Tool

The biggest weakness of TCAL currently is the constraint solver. The time until a satisfactory solution is found could presumably benefit from improving the model, comparing more algorithms, search strategies and trying out different tools or implementing a customized algorithm.

Input and Output

We built a parser to convert our exemplary school's data into TCAL format. With a more adjustable parser, different file formats could be converted into TCAL.

We looked into creating a file that could be imported into the timetabling software that is used by our exemplary school. However, the file would have to include information that is not part of the teacher assignment task, like the class's room or the date when the school year starts and ends. Finding a way to output the assignment into file formats that are used by popular timetabling softwares would be advantageous for integrating TCAL into the normal timetabling work flow.

10 Summary and Conclusion

In this thesis, we designed and implemented a DSL for the assignment of teachers to classes and lessons to support schools in the process of teacher allocation.

Our attention was drawn to this topic by the people in charge of teacher allocation at a local school. They explained to us the difficulties of finding a fair assignment, because so many constraints have to be considered; Regulations have to be obeyed and teachers' or administration's wishes have to be factored in. We could not find a tool or language that would help schools with their teacher allocation and therefore decided to create a language optimized to model and solve the problem. Utilizing the language workbench Spoofax and the constraint solver Choco, we incrementally implemented TCAL (teacher class assignment language).

Using Spoofax simplified the language development process and will facilitate future extension of the language.

We used the domain's terminology and an intuitive grammar to make the language readable and understandable even without programming experience. In contrast to the school's previous storage of data in Excel spreadsheets, the data representation in TCAL is unambiguous and expressive. In TCAL our school's constraints are already built in and can be customized and set as needed. Coding effort is therefore minimized compared to using a general purpose language, and algorithm details are hidden behind a layer of abstraction. The user describes the problem in a declarative manner, rather than having to specify how to solve it.

The compiler of TCAL has multiple phases and converts the AST of a program written in TCAL first into Scala objects and then into a format convenient for the solver. This segmentation simplifies extending the language or exchanging the Choco solver with a different tool in the future.

Real data from our exemplary school was used to test our language. A feasible solution can be found within approximately 30 seconds.

As an alternative, a user can search for a solution that seems best to him by incremental search, which keeps part of a previous solution and only searches for a new allocation of the remaining variables.

TCAL has the potential to help many schools in their process of teacher allocation.

List of Figures

List of Code Listings

References

[1] Patat - international conference on the practice and theory of automated timetabling. http://patatconference.org.

[2] An architecture for workflow scheduling under resource allocation constraints. *Information Systems*, 30(5):399 – 422, 2005.

[3] R. Alvarez-Valdés, F. Parreño, and J. M. Tamarit. A tabu search algorithm for assigning teachers to courses. *Top*, 10(2):239–259, 2002.

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI 2004: 16th European Conference on Artificial Intelligence, August 22-27, 2004, Valencia, Spain : Including Prestigious Applicants [sic] of Intelligent Systems (PAIS 2004) : Proceedings*, volume 16 of *Frontiers in artificial intelligence and applications*, page 146. IOS Press, 2004.

[5] J. A. Breslaw. A linear programming solution to the faculty assignment problem. *Socio-Economic Planning Sciences*, 10(6):227 – 230, 1976.

[6] E. K. Burke, J. H. Kingston, and P. A. Pepper. *Practice and Theory of Automated Timetabling II: Second International Conference, PATAT'97 Toronto, Canada, August 20–22, 1997 Selected Papers*, chapter A standard data format for timetabling instances, pages 213–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[7] M. W. Carter and G. Laporte. *Practice and Theory of Automated Timetabling II: Second International Conference, PATAT'97 Toronto, Canada, August 20–22, 1997 Selected Papers*, chapter Recent developments in practical course timetabling, pages 3–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[8] T. B. Cooper and J. H. Kingston. The solution of real instances of the timetabling problem. *The Computer Journal*, 36:645–653, 1993.

[9] T. B. Cooper and J. H. Kingston. *Practice and Theory of Automated Timetabling: First International Conference Edinburgh, U.K., August 29–September 1, 1995 Selected Papers*, chapter The complexity of timetable construction problems, pages 281–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[10] S. Even, A. Itai, and A. Shamir. On the complexity of time table and multi-commodity flow problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, pages 184–193, Oct 1975.

[11] MetaBorg Software Foundation. Metaborg - metaborg. http://www.metaborg.org/, Accessed May 2nd, 2016.

[12] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005. http://martinfowler.com/articles/languageWorkbench.

[13] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[14] M. Gröbner, P. Wilke, and S. Büttcher. A standard framework for timetabling problems. In *Practice and Theory of Automated Timetabling IV*, pages 24–38. Springer, 2002.

[15] A. Gunawan, K. M. Ng, and K. L. Poh. Solving the teacher assignment-course scheduling problem by a hybrid algorithm. *International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering*, 1(9):491 – 496, 2007.

[16] T. H. Hultberg and D. M. Cardoso. The teacher assignment problem: A special case of the fixed charge transportation problem. *European Journal of Operational Research*, 101(3):463 – 473, 1997.

[17] N. Jussien, G. Rochart, and X. Lorca. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, Paris, France, France, 2008.

[18] L. C. L. Kats and E. Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*, pages 444–463, 2010.

[19] J. H. Kingston. A user's guide to the STTL timetabling language version 1.0, 1999.

[20] J. H. Kingston. *Practice and Theory of Automated Timetabling III: Third International Conference, PATAT 2000 Konstanz, Germany, August 16–18, 2000 Selected Papers*, chapter Modelling Timetabling Problems with STTL, pages 309–321. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[21] J. H. Kingston. The kts high school timetabling system. In *Proceedings of the 6th International Conference on Practice and Theory of Automated Timetabling VI*, PATAT'06, pages 308–323, 2007.

[22] J. H. Kingston. The HSEval high school timetable evaluator, 2012. `http://www.it.usyd.edu.au/~jeff/hseval.cgi`.

[23] J.H. Kingston. *Automated Scheduling and Planning: From Theory to Practice*, chapter Educational Timetabling, pages 91–108. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[24] G. D. P. Konat, L. C. L. Kats, G. H. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *Software Language Engineering: 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, pages 311–331, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[25] G. D. P. Konat, V. A. Vergu, L. C. L. Kats, G. H. Wachsmuth, and E. Visser. The spoofax name binding language. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 79–80, New York, NY, USA, 2012. ACM.

[26] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[27] E. Özcan. *Multidisciplinary Scheduling: Theory and Applications: 1st International Conference, MISTA '03 Nottingham, UK, 13–15 August 2003 Selected Papers*, chapter Towards an XML-Based Standard for Timetabling Problems: TTML, pages 163–185. Springer US, Boston, MA, 2005.

[28] G. Post. Benchmarking project for (high) school timetabling, 2008. `https://www.utwente.nl/ctit/hstt/`.

[29] G. Post, S. Ahmadi, S. Daskalaki, J. H. Kingston, J. Kyngäs, C. Nurmi, and D. Ranson. An XML format for benchmarks in high school timetabling. *Annals OR*, 194(1):385–397, 2012.

[30] C. Prud'homme, J. Fages, and X. Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2015. `http://www.choco-solver.org`.

[31] D. Ranson and S. Ahmadi. *Practice and Theory of Automated Timetabling VI: 6th International Conference, PATAT 2006 Brno, Czech Republic, August 30–September 1, 2006 Revised Selected Papers*, chapter An Extensible Modelling Framework for Timetabling Problems, pages 383–393. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[32] L. P. Reis and E. Oliveira. *Practice and Theory of Automated Timetabling III: Third International Conference, PATAT 2000 Konstanz, Germany, August 16–18, 2000 Selected Papers*, chapter A Language for Specifying Complete Timetabling Problems, pages 322–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[33] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier Science Inc., New York, NY, USA, 2006.

[34] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[35] A. Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127.

[36] G. Solotorevsky, E. Gudes, and A. Meisels. Raps: a rule-based language for specifying resource allocation and time-tabling problems. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):681–697, Oct 1994.

[37] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer. The minizinc challenge 2008-2013. *AI Magazine*, 35:55–60, 2014.

[38] P.I. Tillett. An operations research approach to the assignment of teachers to courses. *Socio-Economic Planning Sciences*, 9(3):101 – 104, 1975.

[39] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[40] E. Visser. Webdsl: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 291–373, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[41] Y. Wang. An application of genetic algorithm methods for teacher assignment problems. *Expert Systems with Applications*, 22(4):295 – 302, 2002.