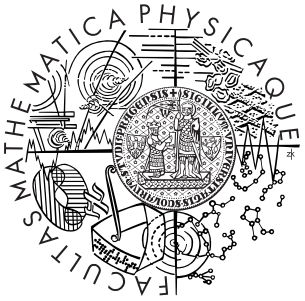
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Michal Tuláček

Constraint solvers

Department of Theoretical Computer Science and
Mathematical Logic

Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Study programme: General Computer Science

2009

I wish to thank my supervisor Dr. R. Barták for advices and patience. My thanks also belongs to my friends V. Kopal, M. Bernát and J. Helmich who helped me with correction of the English text. Last but not least my thanks belong to my coach J. Baxa who clearly stated that failure is not an option and then enforced this policy.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

I declare that I wrote my bachelor thesis independently and exclusively with the use of the cited sources. I agree with lending this thesis.

In Prague, 6th August 2009 Michal Tuláček

Contents

Název práce: Řešiče omezujících podmínek
Autor: Michal Tuláček
e-mail autora: michal@tulacek.eu
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Doc. RNDr. Roman Barták, Ph.D.
e-mail vedoucího: bartak@kti.mff.cuni.cz

Abstrakt: Řešič omezujících podmínek je specializovaný software, určený k řešení problémů popsaných omezujícími podmínkami. Práce podává přehled řešičů a vybrané z nich testuje z pohledu uživatelské přístupnosti a rozsahu problémů, které lze modelovat.

Klíčová slova: omezující podmínky, řešiče, benchmarky

Title: Constraint solvers
Author: Michal Tuláček
Author's e-mail address: michal@tulacek.eu
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Doc. RNDr. Roman Barták, Ph.D.
Supervisor's e-mail address: bartak@kti.mff.cuni.cz

Abstract: Constraint solver is a specialized software used to solve constraint satisfaction problems. The thesis surveys constraint solvers and some of them compares using the criteria of user accessibility and variety of problems which can be modeled.

Keywords: constraint satisfaction problems, constraint solvers, benchmarks

Chapter 1

Introduction

In this thesis we compare several constraint solvers from the perspective of a user who is not experienced in the constraint programming. We focus on easiness of a learning process of each solver and we measure performance by using benchmarks which compare various aspects of examined systems.

1.1 Constraint Satisfaction Problems

Constraint programming is a programming paradigm which uses constraints to describe a solution rather than to program a way of achieving such a solution. The constraint can be any relation which can be asserted as true or false – $X < Y$, Billy is older than Johnny, $Z = 5$, etc. As an example of a problem which can be solved using constraint programming we use the Sudoku puzzle. Sudoku is a worldwide known logical problem which is easy to explain, its difficulty can be scaled and one does not need previous training to solve Sudoku. It makes the problem easier to understand for many people and, therefore, it is very popular. Simple rules of Sudoku are: There is a given table of size nine times nine. Every field of the table contains a number in the range one to nine. In each column all the numbers are different (this enforces that every column contains all numbers in range one to nine). In each row there are also all numbers different. Finally the same rule which applies for columns and rows also restricts three times three sized squares which are in the puzzle marked using bolder lines. The Sudoku is prefilled with a couple of values. These values help at the beginning of solving and the difficulty can be adjusted by their number and placement.

The way how to describe the Sudoku puzzle in constraint programming

Figure 1.1: Example of the Sudoku puzzle

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

is very straightforward. We define the following model:

1. There are 81 variables s_{ij}, $i, j \in \{1, ..., 9\}$ which can contain values in range 1 to 9. We arrange them into a two-dimensional array with size 9×9. (The Sudoku is a table sized 9×9 containing values in range $1, ..., 9$)

2. For all i in $1, ..., 9$ the following condition is true: All values of $s_{i\bullet}$ are different (values in each row are different)

3. For all j in $1, ..., 9$ the following condition is true: All values of $s_{\bullet j}$ are different (values in each column are different)

4. For each square the following condition is true: For all k, l such that k, l are indexes defining the square all values of s_{kl} are different (values in each square are different)

5. For all prefilled values: $q_{mn} = V$ if and only if the field in the column n and row m is prefilled and contains V.

These constraints fully describe the Sudoku puzzle problem and as the reader can see they do not differ from the commonly known rules. A person solving the Sudoku puzzle can use many techniques starting with randomly filling the table and looking if this is a good solution (the algorithm using this technique is called GAT – Generate and Test) and ending with generating all

possible fillings and correcting the solution if something fails (this algorithm is called backtracking). The first approach can miss a correct solution. Since the second approach systematically searches all possible solutions it has to result in a correct solution; however, it can last enormous time to complete it (even on a supercomputer). The secret of a successful solution is in the fact that not all numbers can be filled in a specific field. If there is prefilled value 8 at position [6,7] it means that in row 6 and in column 7 there cannot be another number 8. And because of constraint (4) there also cannot be 8 in the right middle square. A person who does these observations usually writes into the destination field all possible values and as an examination of the puzzle progresses there are less and less possibilities to fill in. In an easy Sudoku after this examination there is at least one field which can be filled with only one number. After filling all such fields the solving continues in the same way until the entire table is filled. A program which uses constraint programming solves it in the same way. For each variable it remembers the range of possible values (we will call it a domain). Before the program starts searching for a solution it tries to eliminate as many values from the domain as possible. It can reveal that the problem does not have a solution (if there is a variable with an empty domain) before a backtracking. It is no surprise that in a user guide to Choco system, one of the available constraint solvers, it is stated "if you know Sudoku, then you know constraint programming." The formal definitions of the constraint satisfaction problem and the solution of the constraint satisfaction problem follows (the definitions are cited from [9]).

Definition 1 *The* **Constraint satisfaction problem** *or* **CSP** *consist of:*

- *a set of variables* $X = \{x_1, ..., x_n\}$*,*
- *for each variable* x_i *a finite set* D_i *of possible values (its* **domain***),*
- *and a set of* **constraints** *restricting the values that the variables can simultaneously take.*

Definition 2 *A* **solution** *to a CSP is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied. We may want to find:*

- *just one solution, with no preference as to which one,*
- *all solutions,*

- *an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.*

All of the constraint problem can be modeled only by using of the binary constraints (the constraints with two variables). The constraint satisfaction problem can be represented as a constraint multigraph where the variables of the model are the nodes of the graph and the constraints over the variables in the model are the edges connecting the appropriate nodes. The edge $e(x, y)$ is consistent if for each value of x there exists a value y such that the constraint is satisfied. As we can see the constraint which is consistent in one direction does not have to be consistent in the second direction.

Definition 3 *The constraint satisfaction problem is* **arc consistent** *if all of the edges in the constraint graph are consistent in the both directions.*

If the problem cannot be transformed to arc consistent state then it obviously does not have a solution; however, the arc consistency itself does not ensure that the problem can be solved. As an example of such a problem let consider the following: We have variables X, Y, Z all with the domain $\{0, 1\}$ with the constraints $X \neq Y$, $Y \neq Z$, $Z \neq X$ on them. The problem is arc consistent. If we examine all constraints there exists for each possible value of first variable a value of second variable which satisfies the constraint and similarly in the second direction. Even though the problem is arc consistent it cannot be solved because there does not exist evaluation of all variables which satisfies the all constraints.

1.2 Constraint programming

We can use many algorithms to solve the constraint satisfaction problems. The most common algorithm consists of two phases which are repeated until the solution is found or we find that there is no solution. The first phase is filtering of the variable domain by eliminating as many as possible values of the domain. We can achieve such filtering by transforming of the problem to the arc consistent state. The filtration phase can end in three ways:

- The domain of some variable is empty and therefore there is no solution for a given problem.

Figure 1.2: The MAC algorithm

```
1  procedure labelling(V,D,C)
2    if all variables from V are assigned then return V
3    select not-yet assigned variable x from V
4    for each value v from Dx do
5      (TestOK, D') := consistent(V,d,C + {x=v})
6      if TestOK = true then
7        R := labelling(V, D', C)
8        if R <> fail then return R
9    end for
10   return fail
11 end
```

- The domains of all variables have only one element. The algorithm found the solution.
- The domains of some variables have more than one element while the problem is arc consistent.

If the problem P is arc consistent but we still do not have the solution we perform the distribution phase. In the distribution phase we introduce a new constraint c. We create two new problems $P \cup \{c\}$ and $P \cup \{\neg c\}$. It is obvious that if there exists a solution of the original problem then at least one of the new problems do have to have the solution. After this phase we run the filtering phase again.

To construct the constraint c used the distribution phase we usually pick a variable x and its value v. The constraint then can be $x = v$ or $x < v$ (but we can use any constraint). By selecting of a proper variable and value we can affect the time needed to compute the solution. The most common way is to pick a variable with the smallest domain because we should be able sooner find out that the variable cannot be evaluated. This strategy is sometimes called *first-fail* strategy. The solvers usually allows the users to choose the strategy or implement their own.

The described algorithm is called *maintained arc consistency* or MAC. Its schematic source code is in the figure 1.2. The code is cited from [7]. More detailed explanations of algorithms used in CSP solving can be found in [9].

The optimization problems can be solved by using of the *branch-and-bound* algorithm. First the algorithm solves the problem P using the standard algorithms. After the problem is solved the value $f(P)$ of the objective function is computed. Then we derive from the problem P a new problem P' with the additional constraint $f(P') < f(P)$ and we run the algorithm

on the new problem. The last found solution which could be satisfied is the expected best solution.

In reading previous paragraphs, a reader could think that the constraint programming is used only as an academic toy for solving Sudoku and for other applications which are useless in a real life. In fact the constraint programming is used in various applications. A few examples of many contain scheduling, an image recognition, financial modeling, planning, vehicle routing, a configuration, computer networks and bio-informatics. The constraint programming was also successfully used at NASA in Deep Space 1 experiment. Deep Space 1 was a space probe using 12 cutting-edge technologies which were never tested in space before. One of these technologies was a remote agent used to plan actions of a space vehicle while only general commands were sent to agent. Agent used a constraint solver for planning [6].

1.3 Constraint solvers

A programmer who wants to solve problems using constraint programming can encode the ideas described in the previous paragraphs or use specialized software, a constraint solver. Constraint solver is a system which uses constraint programming techniques to solve a given problem. There are many solvers available both commercial and freeware. A short list of available systems can be found in the appendix B. A more detailed list is maintained by Roman Barták in the On-line guide to constraints programming at [8].

1.4 Related work

In this thesis we compare several constraint solvers. There exist papers and other work which cover the similar area.

There exists a paper "A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains" by A. Fernandez and others [14] which compares several solvers; however, this paper does not focus on the user experience with the solvers. They compared solvers by their performance on various benchmarks and discussed an implementation of self referential quiz in each solver. We are not benchmarking the same sets of solvers and we do not use similar methodology which means this thesis conclusions cannot fully replace or update the results from [14].

M. Plachá in 2007 wrote the bachelor thesis [18] focused on the same topic as this thesis. The thesis compared SICStus Prolog, ILOG Solver and Gecode/J. She compared the modeling capabilities of these solvers and showed the ways to debug the models. Finally the speed of solvers was measured on several benchmarks. In this thesis we study a wider range of solvers. We used the same criteria as the mentioned thesis by M. Plachá but additionally we compare the accessibility of the solvers for user.

Every year there is held the International Constraint Solver Competition where authors of solvers can compete. The solvers can be submitted in two categories – complete and incomplete. Complete solvers can prove that the instance of a problem is satisfiable or not (or find and prove the optimum). As stated in the rules [21] for each solver there is a *Boolean capability vector* which indicates which constraints the solver can handle. Solvers with the same capability vector can be naturally compared. Solvers with different capabilities can be compared on instances which belong to the intersection of their capabilities, provided it is non-empty. During the competition the solvers are run in the sandbox environment on a Linux cluster. The task is to find a solution of as many benchmarks as possible in the smallest time amount.

There exists a library [15] of the constraint satisfaction problems which can be used in benchmarking and comparing of solver capabilities. The library contains various problems in several fields – optimization problems, combinatorial problems and so on. We encourage the reader to try to implement several problems in the chosen solver as a part of learning of modeling in the solver.

1.5 Outline of the thesis

We described our motivation for this thesis and listed some constraint solvers. In the second chapter we define methodology used to examine some of the mentioned solvers. The examination consists of two parts – performance tests and usability tests. In the third chapter we will define benchmarks used to performance tests. The fourth chapter describes in details each solver, mentions a little from their history but mainly focuses on usability and easiness of learning and using the solver. In the fifth chapter we discuss the performance tests results and compare the solvers. Finally, in the sixth chapter we state a conclusion of the whole examination process.

There are four appendixes to this thesis. The content of the included CD

is in the appendix A. The short list of constraint solvers is in the appendix B. The table of supported constraints is in the appendix C. Finally the source codes of the benchmarks implemented in the compared solvers are in the appendix D.

Chapter 2

Methodology

In the introduction we explained what constraint solvers are and presented several examples of them. In the rest of the thesis we focus on six of them – Mozart/Oz, Choco, Minion, Gecode, ECL^{i}PS^e and SICStus Prolog. The last solver is a professional commercial solution and the others are freely available open source products. The purpose of this thesis is to help new users with choosing of the right solver. Therefore we study complexity of learning and using of each solver and their performance and abilities. We test solvers which use various programming languages and paradigms. The imperative paradigm is represented by a C++ library Gecode and a Java library Choco. Users experienced in logical programming might find interesting SICStus Prolog or ECL^{i}PS^e . Mozart is an implementation of Oz, a multi-paradigm programming language. With these solvers the users can use their current experience and just learn an API of the constraint library. The Minion solver is configured by a solver-specific problem description language. This fact is both advantage and disadvantage. As for disadvantage, we must accept that users cannot use their experience with existing programming languages and have to learn new concepts; however, a specialized language for describing constraint problems can be more accessible for users who do not have any programming experiences but they need to solve the given problem. A general overview of the examination follows.

First, we examine all solvers from the perspective of a user experienced in the given programming language but inexperienced in using of the solver. In case of Minion we expect that the user has general computer knowledge and is able to describe a given problem in constraints. The first examination tries to answer a question how difficult it is to learn to use the solver. We

model problems described in the third chapter and look for constraints which cannot be modelled and we describe possible solutions.

Secondly, a quality of documentation is also an important criterion. A solver can be the best of all, but it is useless if the user cannot understand the usage. The quality of documentation is perceived subjectively and cannot be measured exactly. This means that any evaluation is only informational, although it should be considered. As a documentation we accept a user guide as well as all other available guides, documents, web pages or a doxygen style documentation. An existence of user forums or mailing lists is also an important part of learning of new systems.

Last but not least, we aim for debugging. There are two areas which can be debugged - correctness of the program and correctness of the model. The correctness of the program means that the program does what it should do, that it handles all inputs as the programmer expects and so on. The correctness of model stands for an accurate description of a given problem. The user should be able to inspect variables, visualise a decision tree of search and other information. We discuss the ways how a solver informs about mistakes (and how much descriptive the information is), the tools provided with the solver to debug the program and similarly the tools which can be used to debug the model correctness.

When the user masters the solver and uses it to solve real problems, the time and space efficiency of used algorithms matters. We neither examine source codes, nor analyze time complexity of used algorithms. Instead we measure the time needed to load and the time to solve the problem. If a solver cannot provide such information we measure only a total time. We also measure an amount of consumed memory during the program execution. All measurements are performed several times and averaged to avoid randomness. A robustness test is also performed. In the robustness test we set limit ten minutes and try to determine the length of the longest magic sequence (readers can find a definition of a magic sequence in the section 3.3) can be computed in the given time. We use the models created in the process described in the previous paragraph. In [14] authors have sent models to the solvers' authors and have given them a chance to modify them to achieve the best performance of their solvers. We focus on first-time users of a solver, so we use our own models which are not perfect and, more imporantly, not tuned for any particular solver. All solvers are tested on Debian 4.0 Linux with the kernel 2.6.18 on Pentium 4, 3GHz.

Chapter 3

Benchmarks

In this chapter we shall define benchmarks which will later be used to examine the properties of each solver. We will be dealing with five different cases which are well known and documented: n-queens problem, magic sequence problem, self referential quiz, quasigroup with holes problem and locating warehouses problem. In this chapter, we will show on these benchmarks the different ways of modeling the problem with solvers. The last benchmarking problem – locating warehouses – is an optimization problem. The solver not only has to find correct solutions but also has to evaluate the best of the solutions based on the value of an objective function. For each benchmark a general description is presented as is the formal model of the constraint problem and an implementation in the Essence programming language. The basics of the language are described in the following section.

3.1 Essence programming language

Essence is a programming language for modeling of combinatorial problems. It is easy enough to understand and so simple that even person who has never seen the language before can correctly guess what the expected output of the program is. Every program in Essence consists of three parts. The first part defines the version of the language, the second part the used variables and finally, the third part presents the constraints used on the given variables. These constrained variables can be integers, booleans and vectors or matrices. The language supports sums and loops over the variables; however, the bounds of the sum or loop have to be constant as they are in the process of compiling translated to a sequence of statements. The program itself can be

split into the model definition and parameters definition parts. Examples in this chapters show only the model definition parts. The files containing the parameter definitions can be found on the included CD. The Tailor tool is the compiler which translate the code from the Essence language to the solver specific language. In the current version it can translate the Essence program to Minion input file, FlatZinc and C++ source code which is using Gecode library. Further description of the Tailor system is in the section 4.7.

3.2 N-queens

This benchmark is based on a classic chess task. The player has to place eight queens on the chessboard in a way that none of the queens offends any other. The task can be scaled to a chessboard of any size. The goal is then place n queens onto a table of size $n \times n$ in such way that no queen offends any other. A queen offends all pieces which are placed in the same row, column and diagonal on the chessboard. The problem is a little bit easier if we realize that in order to place n queens on a chessboard with n columns, there has to be one queen per column. Therefore we only need to find out in which rows the queens are in each column. We model a solution of the problem as a vector q_i where $i \in \{1, ..., n\}$. To avoid placing the queens in the same row we simply add constraint that all q_i are different. Finally we have to include the diagonals in the model: two pieces are on the same diagonal, if the difference in the horizontal and vertical coordinates is equal. Therefore we add the constraint $|Q(i) - Q(j)| \neq |i - j|$. Since both the chessboard and the modes of offense the queen can carry out are symmetric, the solutions are symmetric as well. We can decrease the number of solutions if we avoid such symmetries (see the next subsection).

3.2.1 Constraint problem model

- Variables and domains:
  - Positions of queens: $q_1, ..., q_n \in \{1, ..., n\}$, q_i.
- Constraints:
  - All queens are on different rows: $\forall i, j \in \{1, ..., n\} : q_i \neq q_j$,

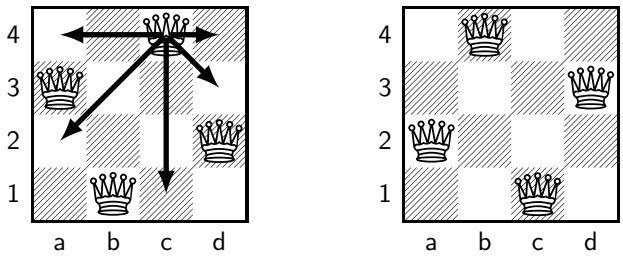
Figure 3.1: Implementation of N-Queens Problem in Essence

```
1 language ESSENCE' 1.b.a
2 find queens: matrix indexed by [int(1..n)] of int(1..n)
3 such that
4
5 alldiff(queens),
6 forall i: int(1..n). forall j: int(i+1..n).
7 | queens[i] - queens[j] | != | i - j |
```

- all queens are on different diagonals: $\forall i, j \in \{1, ..., n\} : |q_i - q_j| \neq |i - j|$
- optional avoiding of the symmetry: $q_1 < q_n$

The Essence implementation (without symmetry breaking) is in the figure 3.1.

Figure 3.2: Solutions of 4-queens problem



3.3 Magic sequence

Magic sequence of length n is a sequence of numbers such as m_i, $i \in \{0, ..., n-1\}$ which satisfies the following condition: The value of m_k is equal to the number of occurrences of value k in the sequence. For example the sequence (2 1 2 0 0) is magic sequence of length five as the condition above is satisfied. The number zero is twice in the sequence and m_0 is equal to two. Similarly one is in the sequence only once and therefore m_1 is one.

Figure 3.3: Implementation of Magic Sequence Problem in Essence

```
1 language ESSENCE' 1.b.a
2 find s : matrix indexed by [int(0..n-1)] of int(0..n)
3 such that
4   forall i : int(0..n-1).
5         ( s[i] = (sum j : int(0..n-1). (s[j] = i)))
```

3.3.1 Constraint problem model

Model for a magic sequence m of length k:

- Variables and domains:
  - Magic sequence items: $m_0, ..., m_{k-1} \in \{0, ..., k\}$.
- Constraints:
  - Value of m_i is i times in the sequence: $\forall i \in 0, ..., k-1 :$ $m_i = \sum_{m_j=i} 1$.

In case that solver does not support the constraint $m_i = \sum_{m_j=i} 1$, we can use an alternative model:

- Values and domains:
  - Magic sequence items: $m_0, ..., m_{k-1} \in \{0, ..., k\}$,
  - auxiliary variables: $\forall i, j \in 0, ..., k-1 : \text{aux}_{ij}$.
- Constraints:
  - $\text{aux}_{ij} = 1$ if and only if $m_j = i$: $\forall i, j \in \{0, ..., k-1\} :$ $(\text{aux}_{ij} = 1) \Leftrightarrow (m_j = i)$,
  - the value of the items of the magic sequence corresponds to the sum of some auxiliary variables: $\forall i \in \{0, ..., k-1\} :$ $m_i = \sum_{j=0}^{k-1} \text{aux}_{ij}$.

The Essence implementation is in the figure 3.3.

3.4 Self-referential quiz

The self-referential quiz is a quiz where the answers to the questions depend on the answers to other questions in the same quiz. There is only one valid answer for each question. A typical question in such a quiz can be:

1. First question where the answer is A:
   (A) 1 (B) 2 (C) 3 (D) 4 (E) there is no question with answer A

2. Answer to this question:
   (A) A (B) B (C) C (D) D (E) E

These quizzes are best modeled using reified constraints. Reified constraint is a constraint in the form $(C \Leftrightarrow x)\&\,(x \in \{0,1\})$. The ways how to construct such quizzes are described in the article by Maja Bubalo [11]. The quiz assignment follows:

1. The first question to which the answer is A:
   (A) 4 (B) 3 (C) 2 (D) 1 (E) none of above

2. The only two consecutive questions with identical answers:
   (A) 3 and 4 (B) 4 and 5 (C) 5 and 6 (D) 6 and 7 (E) 7 and 8

3. The next question with answer A:
   (A) 4 (B) 5 (C) 6 (D) 7 (E) 8

4. The first even numbered question with the answer B:
   (A) 2 (B) 4 (C) 6 (D) 8 (E) 10

5. The only odd numbered question with the answer C:
   (A) 1 (B) 3 (C) 5 (D) 7 (E) 9

6. A question with answer D:
   (A) comes before this one, but not after this one (B) comes after this one, but not before this one (C) comes before and after this one (D) does not occur at all (E) none of the above

7. The last question to which the answer is E:
   (A) 5 (B) 6 (C) 7 (D) 8 (E) 9

8. The number of questions to which the answer is a consonant:
   (A) 7 (B) 6 (C) 5 (D) 4 (E) 3

9. The number of questions to which the answer is a vowel:
   (A) 0 (B) 1 (C) 2 (D) 3 (E) 4

10. The answer to this question is:
    (A) A (B) B (C) C (D) D (E) E

We model the quiz as a table of boolean variables with five columns (A, B, C, D, E) and ten rows, one for each question. The value in the column i and row j is *true* if and only if the answer to the question j is i. Because there is only one answer possible to each question, we constraint the rows of the table to contain only one *true*. The test has only one solution which is showed in the figure 3.1.

Table 3.1: Solution of a Self Referential Quiz

Question	A	B	C	D	E
1	0	0	1	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	1	0	0	0
5	1	0	0	0	0
6	0	1	0	0	0
7	0	0	0	0	1
8	0	1	0	0	0
9	0	0	0	0	1
10	0	0	0	1	0

3.4.1 Constraint problem model

- Variables and domains:
  - The answers to the questions: $s_{1|1}, s_{1|2}, ..., s_{10|4}, s_{10|5} \in \{0, 1\}$.
- Constraints:
  - There is exactly one value 1 in the row:
    $\forall i \in \{1, \ldots, 10\} : \left(\sum_{j \in \{1,\ldots,5\}} s_{i|j}\right) = 1,$

- question 1, A to D:
  $\forall i \in \{1, ..., 4\} : (s_{1|i} = 1) \Leftrightarrow ((s_{4-i+1|1} = 1) \wedge (\forall j \in \{1, ..., 4-i\} : s_{j|1} = 0))$,
- question 1, E:
  $(s_{1|5} = 1) \Leftrightarrow (\forall j \in \{1, ..., 4\} : s_{j|1} = 0)$,
- question 2:
  $\forall i \in \{1, ..., 5\} : (s_{2|i} = 1) \Leftrightarrow (\forall j \in \{1, ..., 5\} : s_{3+i-1|j} = s_{3+i|j})$,
- question 3:
  $\forall i \in \{1, ..., 5\} : (s_{3|i} = 1) \Leftrightarrow ((s_{4+i-1|1} = 1) \wedge (\forall j \in \{4, ..., 2+i\} : s_{j|1} = 0))$,
- question 4:
  $\forall i \in \{1, ..., 5\} : (s_{4|i} = 1) \Leftrightarrow ((s_{2i|2} = 1) \wedge (\forall j \in \{1..i-1\} : s_{2j|2} = 0)$,
- question 5:
  $\forall i \in \{1, ..., 5\} : (s_{5|i} = 1) \Leftrightarrow (s_{2i-1|3} = 1)$
- question 6, A:
  $\forall i \in \{1, ..., 5\} : (s_{6|1} = 1) \Leftrightarrow (\exists j \in \{1, ..., 5\} : s_{j|4} = 1 \wedge$
  $\forall j \in 7, ..., 10 : s_{j|4} = 0)$
- question 6, B:
  $\forall i \in \{1, ..., 5\} : (s_{6|2} = 1) \Leftrightarrow (\exists j \in \{7, ..., 10\} : s_{j|4} = 1 \wedge$
  $\forall j \in 1, ..., 5 : s_{j|4} = 0)$
- question 6, C:
  $\forall i \in \{1, ..., 5\} : (s_{6|3} = 1) \Leftrightarrow (\exists j \in \{1, ..., 5, 7, ..., 10\} : s_{j|4} = 1)$
- question 6, D:
  $\forall i \in \{1, ..., 5\} : (s_{6|4} = 1) \Leftrightarrow (\forall j \in \{1, ..., 10\} : s_{j|4} = 0)$
- question 6, E:
  $\forall i \in \{1, ..., 5\} : (s_{6|5} = 1) \Leftrightarrow (s_{6|4} = 1)$
- question 7:
  $\forall i \in \{1, ..., 5\} : (s_{7|i} = 1) \Leftrightarrow (s_{i+4|5} = 1) \wedge (\forall j \in \{i+4+1, ..., 10\} : s_{j,5} = 0))$
- question 8:
  $\forall i \in \{1, ..., 5\} : (s_{8|i} = 1) \Leftrightarrow \left(\sum_{j=1}^{10} \left(s_{j|2} + s_{j|3} + s_{j|4}\right) = 7 - i + 1\right)$
- question 9:
  $\forall i \in \{1, ..., 5\} : (s_{9|i} = 1) \Leftrightarrow \left(\sum_{j=1}^{10} \left(s_{j|1} + s_{j|5}\right) = i - 1\right)$.

Figure 3.4: Implementation of the Self Referential Quiz in Essence

```
language ESSENCE' 1.b.a
find s : matrix indexed by [int(1..10), int(1..5)] of bool
such that
$ there is only one answer to each question and there is not any unanswered question
 forall row : int(1..10). ((sum col : int(1..5). s[row,col]) = 1),
$ Question 1
$ A to D
  forall col : int(1..4). ( (s[1,col] = 1) <=> ( (s[(4-col+1),1] = 1) /\ ( forall row :
      int(1..(4-col)). (s[row,1] = 0) ) ) ),
$ E
  (s[1,5] = 1) <=> (forall row : int(1..4). (s[row,1] = 0)),

$ Question 2
  forall col : int(1..5). ( (s[2,col] = 1) <=> ( forall col2: int(1..5). (s[3+col-1,
      col2] = s[3+col,col2]) ) ),
$ Question 3
  forall col : int(1..5). ( (s[3,col] = 1) <=> ( (s[(4+col-1),1] = 1) /\ ( forall row :
      int (4..2+col). s[row,1] = 0 ) ) ),
$ Question 4
  forall col : int(1..5). ( (s[4,col] = 1) <=> ( (s[col*2,2] = 1) /\ ( forall row : int
      (1..(col-1)). s[row*2,2] = 0 ) ) ),
$ Question 5
  forall col : int(1..5). ( (s[5,col] = 1) <=> (s[2*col-1,3]=1) ),
$ Question 6
  (s[6,1] = 1) <=> ( ( exists row : int(1..5). s[row,4] = 1 ) /\ ( forall row : int
      (7..10). s[row,4] = 0 ) ),
  (s[6,2] = 1) <=> ( ( exists row : int(7..10). s[row,4] = 1 ) /\ ( forall row : int
      (1..5). s[row,4] = 0 ) ),
  (s[6,3] = 1) <=> ( ( exists row : int(7..10). s[row,4] = 1 ) /\ ( exists row : int
      (1..5). s[row,4] = 1 ) ),
  (s[6,4] = 1) <=> ( forall row : int (1..10). s[row,4] = 0 ),
  (s[6,5] = 1) <=> (s[6,4] = 1),
$ Question 7
  forall col : int(1..5). ( (s[7,col] = 1) <=> ( (s[col+4,5] = 1) /\ ( forall row : int
      (col+4+1..10). s[row,5] = 0 ) ) ),
$ Question 8
  forall col: int(1..5). ( (s[8,col] = 1) <=> ( ( sum row: int(1..10). (s[row,2] + s[
      row,3] + s[row,4]) ) = (7-col+1) ) ),
$ Question 9
  forall col: int(1..5). ( (s[9,col] = 1) <=> ( ( sum row: int(1..10). (s[row,1] + s[
      row,5]) ) = (col-1) ) )
$ Constraints for question 10 are useless
```

The Essence implementation is in the figure 3.4.

3.5 Quasigroup with holes

Quasigroup or latin square is a table of size $n \times n$ filled with numbers in the range $1, ..., n$ such that all values in each row and in each column are unique. There can be also additional condition on the items of quasigroup, for example that the items on the main diagonal have to be even. The task is to fully fill the given partly filled quasigroup. The completed quasigroup has to satisfy all previously stated constraints. This problem is called a *quasigroup completion problem* or QCP. Unfortunately, this benchmark does not provide a consistent result. Some partial fillings can be solved surprisingly easily while other can be extremely demanding; some can even be impossible

to solve. The crucial problem is that determining whether the problem can be solved or not is a NP-complete task [5]. Therefore we cannot determine with certainty whether the problem is only too hard for the solver or the solution does not exist at all and the solver has to search through enormous state space. To avoid this uncertainty, we use a modification of QCP called *quasigroup with holes* or QWH. First we generate a fully filled quasigroup which satisfies the given conditions and then we exclude some of the values; this quasigroup with holes is the new assignment for QCP. We have a guarantee that the assignment is correct and that it can be solved. Generation of QWH assignments was studied by D. Achlioptas et al. who found out that the difficulty of finding a solution of such problem depends on the size of the so called backbone [5]. The backbone is a set of positions in the quasigroup which have the same value in all solutions. If the size of the backbone is close to 0%, there are many different solutions and the solver can find some "by accident". On the other, hand if the backbone is close to 100%, there is only one solution and all constraints lead towards it. The experiments [5] showed that this interesting behavior is somewhere near the 30%. They also showed that the hard quasigroups of the order n have $1.6n^{1.55}$ holes.

The quasigroups we use have no additional conditions. The used quasigroups are produced by the generators *lsencode* developed by Carla Gomez and *walksat* by Henry Kautz.

3.5.1 Constraint problem model

Model for a quasigroup of the order n. The assignment values are in the vector data_{ij}:

- Variables and domains:
  - Quasigroup items: $q_{11}, ..., q_{nn} \in \{1, ..., n\}$.
- Constraints:
  - All items in one row are unique: $\forall i \in \{1, ..., n\} : \forall j, k \in \{1, ..., n\} : q_{ij} \neq q_{ik}$,
  - all items in one column are unique: $\forall i \in \{1, ..., n\} : \forall j, k \in \{1, ..., n\} : q_{ji} \neq q_{ki}$
  - some items of the quasigroup are preassigned: data_{ij} defined $\Leftrightarrow (q_{ij} = \text{data}_{ij})$

Figure 3.5: Implementation of Quasigroup With Holes Problem in Essence

```
1 language ESSENCE' 1.b.a
2 letting nDomain be domain int(1..n)
3 find qcp : matrix indexed by [nDomain,nDomain] of nDomain
4 such that
5   forall i : nDomain. alldiff(qcp[i, nDomain]),
6   forall i : nDomain. alldiff(qcp[nDomain,i])
```

The Essence implementation is in the figure 3.5.

3.6 Locating warehouses

Let us assume that we want to help a hypothetical business company with the decision which warehouses should be built for their stores and which warehouses should supply which store. The main criterion is in this case the cost of the solution. The cost has two components. The first one is payment of the constant cost for opening a new warehouse. The second component is the price for the distribution of goods from a warehouse to the store. The price varies and is different for all pairs (warehouse, store). Each possible warehouse has a defined maximum capacity which accounts for the number of stores which can be supplied from this warehouse. As the last condition of the model, we state that all stores have to be supplied. Our task is to choose the solution with the minimal total cost.

Both maximum capacities of warehouses and a table with prices for supplying each of the pairs (warehouse, store) are the required input parameters for the solver which computes the vector s_i, $i \in \{1, ..., \#\text{ of stores}\}$ where s_i indicates which warehouse supplies the store i.

3.6.1 Constraint problem model

We can build W warehouses. The price for opening of a new warehouse is fixed and stored in the parameter openCost. We also have S shops. The maximum capacity of warehouses is given by vector w, where w_i is the maximum number of shops which can be supplied by the warehouse i, $i \in \{1, ..., W\}$. Finally, we have a matrix of supply costs supplyCost where supplyCost_{ij} translates as the cost of supplying the shop j from the warehouse i.

- Variables and domains:
  - The total cost – the value of an objective function: $\text{totalCost} \in \mathbb{N}$,
  - the number of opened warehouses: $\text{numberOpen} \in \{0, W\}$,
  - the indication whether the warehouse is open: $\text{open}_1, ...,$ $\text{open}_W \in \{0, 1\}$,
  - the indication which warehouse supplies which store: $\text{supplier}_1, ...,$ $\text{supplier}_S \in \{1, ..., W\}$,
  - the supply cost for given store: $\text{cost}_1, ..., \text{cost}_S \in \mathbb{N}$,
  - total supply cost: $\text{costSum} \in \mathbb{N}$.
- Constraints:
  - The objective function: $\text{totalCost} = \text{costSum} + \text{numberOpen} \cdot$ openCost,
  - the total supply cost: $\text{costSum} = \sum_i \text{cost}_i$,
  - the number of opened warehouses: $\text{numberOpen} = \sum_i \text{open}_i$,
  - the maximal capacity of each warehouse: $\forall i \in \{1, ..., W\}$: $w_i \geq \sum_{\text{supplier}_j = i} 1$,
  - the warehouse is open if it supplies at least one shop: $\forall i \in \{1, ..., W\} : (\text{open}_i = 1) \Leftrightarrow \left(\left(\sum_{\text{supplier}_j = i} 1\right) > 0\right)$,
  - the supply cost computation: $\forall i \in 1, ..., S, \forall j \in \{1, ..., W\}$: $(\text{supplier}_i = j) \Rightarrow \left(\text{cost}_i = \text{supplyCost}_{ij}\right)$.

The Essence implementation is in the figure 3.6.

Figure 3.6: Implementation of Locating Warehouses Problem in Essence

```
language ESSENCE' 1.b.a

given Capacity : matrix indexed by [WarehousesRANGE] of int(0..numberOfStores)
given StoreWarehouseCost : matrix indexed by [StoresRANGE,WarehousesRANGE] of CostRANGE
letting CostRANGE be domain int(0..maxCost)
letting StoresRANGE be domain int(0..numberOfStores-1)
letting WarehousesRANGE be domain int(0..numberOfWarehouses-1)

find
  TotalCost : CostRANGE,
  Open : matrix indexed by [WarehousesRANGE] of int(0..1),
  NumberOpen : int(0..numberOfWarehouses),
  Supplier : matrix indexed by [StoresRANGE] of WarehousesRANGE,
  Cost : matrix indexed by [StoresRANGE] of CostRANGE,
  SumCost : CostRANGE

minimising TotalCost

such that
  TotalCost = SumCost + NumberOpen * warehouseCost,
  SumCost = sum j : StoresRANGE. (Cost[j]),
  NumberOpen = sum j : WarehousesRANGE. (Open[j]),

  forall i : WarehousesRANGE.
  (Capacity[i] >= (sum j : StoresRANGE. (Supplier[j] = i))),

  forall i : WarehousesRANGE.
  (((sum j : StoresRANGE. (Supplier[j] = i)) > 0) => (Open[i] = 1) ),

  forall i : WarehousesRANGE.
  (((sum j : StoresRANGE. (Supplier[j] = i)) = 0) => (Open[i] = 0) ),

  forall i : StoresRANGE. forall j : WarehousesRANGE. ( (Supplier[i] = j) => (Cost[i] =
      StoreWarehouseCost[i,j]) )
```

Chapter 4

Constraint solvers

In this chapter we describe the constraint solvers. The description of the solvers is based on the author's observation and the information available in the documentation of each solver. The reference to the documentation of each solver is in the *subjective description* section of each solver.

4.1 Mozart/Oz

Mozart is an implementation of the multi-paradigmatic language Oz. Oz is a functional language with built-in support for threaded applications and paralelisation. It also contains support for constraint solving. Mozart/Oz can run the subroutines on computers connected to the cluster. Since it is a multi-paradigmatic language the user can write imperative as well as Prolog-like programs. The language also offers classes including inheritance and creating the objects. The language was designed for the highest variability of usage because the programmer can use all the features of imperative, functional, logic, object-oriented and other paradigms in one program. In the standard distribution it is shipped as a standalone compiler which compiles into native executable code. Moreover it can run in the interactive mode. The programmer can feed the compiler with a single line, buffer or a whole program and the compiler immediately responds. As an IDE Mozart standardly uses the EMACS system.

Just like in any other functional language the programmer can assign to the variable only once per its lifetime. Therefore all variables holds also its state. In case that an operation is performed over the not assigned variable the actual command is suspended until the problem is resolved. This means

that after executing the following code the variable c will contain 5:

```
a = 5
if a > b then c = 5 else c = 6
b = 4
```

4.1.1 Solver description

As stated in the previous section the language has integrated a constraint solver. The solver can solve problems with variables whose domains are finite sets. A finite domain set can contain the natural numbers including zero. The maximal value of variable is limited and is smaller than the maximal integer. The computation model for constraint propagation is called a *space*. The space consists of several propagators connected to a constraint store. The constraint store contains conjunction of ground constraints. Ground constraints are constraints in the form $x = n$ or $x \in D$. For example the constraint store could contain the constraint $x = 6 \wedge y \in \{1, ..., 12\} \wedge z = y$. Propagators contain other constraints, for example $x > y$ or $a^2 + b^2 = c^2$. Propagator for a constraint c is an independent agent which tries to shrink the domain of variables constrained by c. A solution is such assignment of values to variables which satisfies all the conditions in propagators.

Example 1 *Let us have variables X and Y and the following constraints: $X \in \{0..9\}$, $Y \in \{0..9\}$, $X + Y = 9$, $2X + 4Y = 24$.*

1. *The constraint store contains: $X \in \{0, ..., 9\}$, $Y \in \{0, ..., 9\}$. Propagators: $X + Y = 9$ a $2X + 4Y = 24$.*

2. *The first propagator cannot do anything but the second changes the constraint store to $X \in \{0, ..., 8\}, Y \in \{2, ..., 6\}$.*

3. *The first propagator changes the constraint store to $X \in \{3, ..., 7\}$, $Y \in \{2, ..., 6\}$.*

4. *The second propagator changes the constraint store to $X \in \{4, ..., 6\}$, $Y \in \{3, ..., 4\}$.*

5. *The first propagator changes the constraint store to $X \in \{5, ..., 6\}$, $Y \in \{3, ..., 4\}$*

6. *The second propagator finally changes the constraint store to* $X = 6$, $Y = 3$.

Propagation can be either interval or domain. The interval propagation changes only the bounds of domain. Domain propagation also eliminates the values of the domain. The domain propagation is on the first sight better technique but is more complex than the interval propagation. Therefore the interval propagation is more frequently used.

After propagation if the system is in a stable state and still the solution was not found the distribution phase begins. Mozart choose a variable x and value v from the domain D_v and create two new spaces $S \cup \{x = v\}$ and $S \cup \{x \neq v\}$. The computation then continues with the propagation phase in the new spaces. If the propagation phase ends with a failure the space also fails. The problem has no solution if all its spaces have failed.

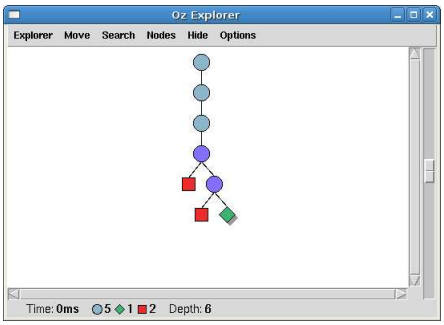
We can choose from several distribution strategies. Choosing of the proper strategy noticeable affects the computation time. For most problems the first-fail strategy is the most suitable. However the user can implement his own distribution strategies to fully suit his needs.

Two techniques can be used in the solving of the optimization problems. The naïve technique introduces auxiliary variable o and adds the constraint $o = f(P)$ where f is the objective function. Then the variable o is increased until the solution is found. The second possible technique is the branch-and-bound algorithm which is described in the section 1.2.

4.1.2 Debugging support

Mozart/Oz offers to the user the interactive tool Explorer. The Explorer can be used to explore the search tree including the choice nodes. The user can use the Explorer in the interactive mode and choose the subtrees of the search tree to be expanded. The Explorer tool screenshot is in the figure 4.1. The circles denote the choice nodes of the tree, the diamonds mean the solution of the problem and finally the squares are the branches with no solution. The lighter color denotes the nodes which can be expanded. On the figure there is one solution, two unsuccessful branches and five choice nodes. Three of the choice nodes can be still expanded. The Explorer tool offers also exporting of the tree diagram to PostScript.

Figure 4.1: The Explorer tool



4.1.3 Subjective description

The Mozart/Oz system is well documented. There is available a website [1] where the reader can find the documentation for the whole system. The difficulty level of the documentation varies from the tutorials and basic documentation to the advanced topics. The Oz language was new for the author of this thesis but the documentation was sufficient to understand the programs written in Oz and to be able to write new programs by himself. The development of the system seems to be stopped now. The last version was released a year ago. The release contains a bug which results in a problems while the compiling of the Mozart. There has been nothing done about it even though the authors stated that the bug will be corrected.

4.2 Choco

Choco is a constraint solver which is implemented as a library in Java programming language. It is distributed as a JAR package having a JavaDoc documentation included. It is quite easy to install it even for beginners in Java and it lasts about five minutes in commonly used IDEs. Since the Java is used, the Choco solver is available for various platforms and operating systems. As far as it is not our goal to describe possibilities of the host environment, we are not about to further discuss the Java features. Choco is being developed at Ecole des Mines de Nantes in France and it is freely available for download from the SourceForge. The main number of the current version is 2. Choco divides the problem solution into two parts – a model

and a solver itself. The model contains variables and constraints given in the problem. Afterwards, the solver is given the model as an input and it tries to find a solution. Variables in the model can be integers, real numbers or sets. Then the solver is able to find a solution for the current model. A user can get information from the solver whether the problem has a solution or it contains a conflict. There is an interface for resolving solutions themselves, whereby one can ask for the first, the following or all existing solutions. If we define a variable equal to a value of an objective function, the solver can either minimize or maximize this variable . Furthermore, the solver allows us to choose a strategy which might perfectly fit the given problem. The variables of the solver depends on the variables of the model and one can resolve the values only through the variables of the solver.

4.2.1 Solver description

As it has been already mentioned in the previous section, the problem solving is divided into two separated tasks – to define a model and to deploy the model to a well-configured solver. The model as well as the solver are the independent Java objects. First we describe the model and after that we look at the solver.

Model

The model is an instance of the class CPModel. In Choco the variables of the model are represented as objects of the following types: IntegerVariable, RealVariable and SetVariable. Those variables, generally, are not created using the keyword new, but in the Choco there are factory methods for this purpose. One has to register those variables first by calling a function `CPModel::addVariable`, or `CPModel::addVariables` when adding an array of variables at once. While registering variables into a model, we can set additional properties to the variables, for example to set whether it is a decision variable, or a variable containing a result of an objective function. It is not necessary to set those properties every time; however, they might rapidly improve the computation time. Alternatively, we can define those properties in the solver, which way is described later in the following section.

Once we have registered the variables, a definition of constraints follows. It is possible either to use a large number of build-in constraints, or to define our own constraints. Some of the constraints which are available natively in the solver are listed in the appendix C. Each constraint

fits in one of the following groups: basic constraints (true, false, relation operators), basic expressions (goniometric functions, powers, sums), other constraints (`abs`, `div`, `max`, ...), reified constraints (`and`, `or`, `ifOnlyIf`) and global constraints (`allDifferent`, `occurenceMax`, ...). Furthermore, there are constraints available which might be used for modeling geometric constraints, scheduling constraints and constraints for a sequence of variables which is accepted by a finite automaton.

Apart from the build-in constraints, is it possible to define own constraints. The first step is to define a constraint $p(x, y)$ as a set of the compatible values (a, b), where p is satisfied if $x = a$ and $y = b$, or, eventually, as a set of the incompatible values. In that case the set is defined as a table of the values. Besides, we can define the constraint as a predicate, which has to be satisfied, whereby the constraint is an instance of a class derived from a class BinRelation with a method checkCouple having implemented. This function takes two values as parameters and returns boolean value whether the condition was satisfied or not. Similarly, we can define constraints over tuples. For all such constraints (either binary or tuple) we can specify the desired algorithm for arc consistence. There are AC3, AC2001, AC3rm and AC3 algorithms available for the binary constraints and AC32, AC3rm, AC2001 and AC2008 algorithms for the tuple constraints. These algorithms are the variants of the original AC algorithm. Further description for some of them can be found in [9].

As we found out while implementing of the Self Referential Quiz the `ifOnlyIf` constraint is implemented as an extensional constraint (the table of compatible values is computed). If the constrained variable has a large domain then the table could be larger than the possible memory.

Solver

A solver is an instance of class CPSolver, which tries to find a solution according to the model from the previous section. The solver starts with reading the variables of the model and converting them into variables of the solver (IntegerVariable into IntDomainVar, RealVariable into RealVar and SetVariable into SetVar). Afterwards, it reads the constraints of the model and creates constraints of the solver according to them. Then the solver uses a search strategy and searches for solutions. Since the chosen strategy is a key factor for the speed of solving, one can configure its various options. A user can specify a selector and an iterator. The selector specifies which

variable is about to be taken in the next solver's decision and the iterator chooses each of available values and iterates over them. In a standard distribution of Choco there are basic selectors such as *variable with a minimum domain*, *variable with a maximum domain* and so on. The iterators can try values in ascendant or descendant order. An alternative to iterator is a value selector, which returns next available value when required. As for value selector, we can use, for example, the minimal value in a domain, a random value in a domain and so on. We can choose different user-defined strategies for various groups of variables so as to follow the specified problem in the best way. In that case we define the solver's behavior through so-called goals. A goal contains a definition of a strategy, that means a selector for certain variables and an iterator over values.

Solving large-scale problems might be enormously time demanding, take too much system resources and so on. To avoid this we can define solver limits. In the solver we can set a time limit, a limit for a number of nodes, a depth of backtracking, a number of fails or a limit for CPU time. Apart from that, the users can define their own limits.

Once the solver has read the model and the strategies are defined, it starts solving the problem. The solver offers an interface for accessing either each solutions (solve, nextSolution), or to get all the solutions at once. Moreover, we can specify a variable which the solver tries to minimize or maximize. Since the result is held in variables of the solver and not in user-defined variables of the model, it is required to resolve the solver's variables by calling a function CPSolver::getVar, which accepts a variable of a model and returns a variable of a solver.

4.2.2 Debugging support

Choco does not include any tools for the graphic visualization of the search tree such as the systems Mozart or Gecode do; however, in the Choco it is possible to print out a log of the solving process. One can configure several levels how detailed information is logged varying from nothing to a complete list of what the Choco does internally.

4.2.3 Subjective description

The system has a good documentation [2], although it is a little bit bad organized. Even though the careful readers find virtually everything they

look for. A documentation for developers is generated by JavaDoc system. Due to that fact it is available as a hinting tool for many users of common Java IDEs, that definitely helps for better understanding of the solver. Since the development of the solver is maintained at SourceForge server, it is quite easy to access source codes as well as a history of versions via revision control system Subversion. One can find there also a technical support forum, where the authors answer the users' questions. The reaction time is very low and the answers are of high quality so most of the problems are quickly fixed.

4.3 Minion

Minion is a constraint solver which works as a standalone application. Minion takes as an input the problem description and returns the solutions if there are any. Minion is open source and available at SourceForge. The problem description is contained in a file with a special format. The file contains definitions of variables, their domains and constraints on them. The best description of the input format could be "the constraint assembly language". The constraints cannot use as their parameters expressions. If we need to constrain the expression we have to introduce a new variable to be equal to the expression and then constrain this variable. For example let us assume we can use constraints $X = |Y|$ and $X < Y$ and we need to use the constraint $X < |Y|$. We have to introduce the auxiliary variable a and post two constraints $a = |Y|$ and $X < a$. Using of the many expressions in our problem leads to the enormous number of auxiliary variables. Moreover the language does not contain loops. Therefore we cannot post a various number of constraints based on the size parameter of the model. We have to state all constraints for a given problem instance. As a result the definition of 4-queens problem cannot be easily transformed to 8-queens problem. The set of constraints is limited and cannot be extended. Moreover some basic constraints have to be used in an unusual way. If we want to constrain $a = \sum w_i x_i$ we have to use a pair of constraints $a \leq \sum w_i x_i$ and $a \geq \sum w_i x_i$ since the variant with equal sign is not in the system. The used variables can be booleans, bounded integers, discrete integers and sparse bounded integers. Bool variable is a variable with domain $\{0, 1\}$. Bounded integer variable is an integer variable where the solver stores only its bounds. Discrete integers are generally the same as bounded integers except they can contain holes in the interval. Finally sparse bounded integers are variables which have a small number of non consecutive values specified in the file.

During the computation only bounds of the spare bounded integers are updated.

The input format is not very human-friendly. Even for small problems the number of auxiliary variables grows over reasonable limits and the input is not easy to understand. It is a reasonable solution to use a generator to generate a Minion input file. User can either write his own ad hoc generator for his problem or use a tool like Tailor which is described in the section 4.7.

4.3.1 Solver description

As stated in the previous section Minion is a standalone executable which takes as a parameter filename of the input file or through the standard input the contents of input. It automatically starts computation and writes on the standard output or into the specified file solutions. It can handle only integers and therefore the problems have to be encoded in a such way. The format of the input file is not stable but uses an identifier which prevents from misinterpretation of the input. If the user feeds the Minion with an unsupported format of the file the solver informs about that and ends. The current version of the input file is 3, and the user can recognize it by observing the first line of the input, which has to contain only string `MINION 3`. After identification line there follows the sections of the file. The file can contain the following sections:

- Definition of the variables
- Definition of the tuples
- Definition of the constraints
- Definition of the search parameter

The list of supported constraints can be found in the appendix C. The search engine behavior can be adjusted either in the input file or at the command line by parameters. The user can adjust the order of variables during the selection phase of search, decide whether to find all solutions or only one or how to format the output.

In the section D.3 we included the implementation of the Magic Sequence benchmark of the length five. As the reader can see the input file is divided into several sections. The specification of the file format comes on the first line. Then follows the sections of the file denoted by the label `*** SECTION`

`NAME ***`. These sections can be in the file in any order and repeated as many times as needed. The only exception is the `EOF` section which denotes the end of the file and should be in the file only once at the end of the file. We used the alternative model of the problem as described in the section 3.3. Therefore we need five variables for the sequence and twenty-five auxiliary variables. We define them in the `VARIABLES` section on lines 4 – 30. Then the constraints are defined in the `CONSTRAINTS` section. The implication is modeled using the reification constraint. Since the Minion have only the constraints $\sum x \leq a$ and $\sum x \geq a$ but does not have the constraint $\sum x = a$ we have to use the constraints `sumleq` and `sumgeq`. Finally the `SEARCH` section defines the order of the variables. The variables will be searched in that order. The `SEARCH` section defines also the solution using the `PRINT` command. All variables marked for printing will be printed out as a solution. To generate the Minion input files we used the Tailor tool which compiled the Essence source code into the Minion input file; however, the large models were problematic for this tool. Therefore we had to write for some benchmarks the ad-hoc generators of the input file.

4.3.2 Debugging support

Minion offers printing out the search tree where one can observe the solvers actions. However some other output than this printout is not available. Since the modeling in Minion is difficult the user will probably use the Tailor tool. Tailor offers not only the translation to the Minion format but direct invocation of the Minion, passing the input to it and show the solution. The user then can directly modify the Essence source code and perform the invocation again if the results are not what he expected.

4.3.3 Subjective description

The largest problem of the solver is the limited set of constraints. If the problem needs more sophisticated constraints then we cannot represent it in the Minion. The second large problem is the input format. The problem can be resolved by using the Tailor tool; however, for the large problems the tool is not sufficient. Apart of that as a standalone executable which can be called from the command line it can bring the capability of the constraint solving to programs written in the languages like bash. The only thing which is needed is the capability to assemble the input file in the proper format.

There is available the reference guide [17] with description of all constraints available and description of the input format.

4.4 Gecode

Gecode is a C++ library for solving constraint problems. It allows to model a problem which contains integers, boolean variables and finite integer sets. Gecode is a free open source software just like most of the mentioned solvers are. The library is distributed in the source codes and for the Microsoft Windows there is also an installer with precompiled libraries. Optionally, an user needs a Qt library, which is used in a visualization graphic tool Gist. Apart from the basic constraints, Gecode has also constraints for scheduling, finite automatons, graphs and so on. We describe those constraints further in the following section. The key person behind the Gecode solver is Christian Schulte, who also participated in a development of the system Mozart/Oz.

4.4.1 Solver description

A problem is modeled as a class inherited from a class Space. In this particular class there are defined variables and constraints. The variables are objects of one of the following types: IntVar for integers, BoolVar for booleans and SetVar for finite integer sets. Compared to the other solvers the boolean variables aren't just integers with a domain $\{0, 1\}$. It is not even allowed to declare a constraint $b = i$ having a boolean variable b and an integer i. When a relation between the boolean and integer variables is required, one can use either reified constraints $(i = 1) \Leftrightarrow (b = true)$, or channelling constraints $(b_j = 1) \Leftrightarrow (i = j)$ (such as $b = (0, 0, 0, 1, 0, 0)$, $i = 3$). The constraints are global functions in a Gecode namespace. To add a constraint to the model one has to use the `post` function which takes a constraint as a parameter. Due to the function overriding, most of the constraints are implemented in a way, where there are more variants each available through the overriding; and the appropriate one is chosen. If the users uses a so-called *minimodel*, they can use short expressions, for instance a constraint $a = x \cdot y$ can be written as `post(home, a == x*y )` instead of `mult(home,x,y,a)`; however, the minimodel works only as a "syntactic sugar" and, therefore, it does not come up with any new constraints and it only ease a way how the current ones are written.

Apart from the standard arithmetic constraints and order constraints, Gecode offers wide range of specialized constraints. For SAT there is a constraint `clause(space,rel,x,y,z)`, which is equal to a condition $\bigvee x_i \vee \bigvee \neg y_i = z$, respectively, $\bigwedge x_i \wedge \bigwedge \neg y_i = z$ depending on a parameter *rel*. There is another interesting constraint *count*, which guarantees that $\#\{i|x_i = j\} = y_j$ is satisfied. This brings us to the fact that the benchmark Magic sequence can be modeled and implemented by using only one constraint `count(*this, x, x)`. Moreover, there are also available the extensional constraints. Those are constraints defined using of the extension, for example, a deterministic finite automaton. In such a case, the constraint has the following pattern: *x is a word, which is accepted by the automaton A*. As for graph constraints, Gecode has a constraint *x is a Hamiltonian cycle*. For an area of scheduling, there is available a constraint `cumulatives(home, resource, start, duration, end, height, limit, atmost)`, which guarantees, that in a set of tasks T (task $T_i = \langle \text{machine}_i, \text{start}_i, \text{duration}_i, \text{end}_i, \text{height}_i \rangle$) will be performed on the available machines. Each machine can handle at most limit_j tasks at one moment.

The following constraints can be used for constraining of the set variables: standard set operations, constraints for cardinality of set and a constraint which guarantees, that a weighed sum of set has a given value. That means if we have a set x, a weight vector w and the total should be y, the constraint has a pattern $\sum x_i w_i = y$. Furthermore, there are two constraints available, both of which are related to convexity. A set is convexed, if it contains a continual interval of numbers. A set $\{1, 2, 3\}$ is convex, but $\{1, 3, 4, 5\}$ is not convex, since it does not contain the number 2. The convex hull is the least convex superset. A constraint `convex(home,x)` guarantees, that x is convex and `convex(home,x,y)` means, that y is a convex hull for x. If one needs to model an optimization problem, the problem representing class is derived from a class MaximizeSpace (or MinimizeSpace) instead of a class Space. In the class there has to be implemented the `cost` function which returns the value of the objective function.

Searching is maintained by a function `branch(home,x,var,val)`, which sets a search vector x. While distributing, a parameter *var* defines an algorithm for choosing a variable and a parameter *val* defines an algorithm for choosing of a value of variable on which depends the distribution. The following code shows the call interface for the application to invoke a search for the solution:

```
Model* m = new Model;
```

```
SEARCH<Model> e(m);
delete m;

while (Model* s = e.next())
{
s->print();
delete s;
}
```

A model is a class derived from a class Space or, respectively, MaximizeScript, which represents a model of CSP. In the real application the `SEARCH` must be changed to one of the followings: `DFS` – the depth-first left-most search, `LDS` – the limited discrepancy, `BAB` – the branch-and-bound algorithm and finally `Restart` – the depth-first left-most restart search. Both algorithms `BAB` a `Restart` can be used when searching an optimal solution. All of the algorithms can be used for a parallel computation. Having a computer with more CPUs, one can reach much faster computation.

4.4.2 Debugging support

In Gecode there is a tool Gist, which visualizes a search tree. It is a tool quite similar to the tool Explorer in Mozart/Oz solver. The interface, graphic symbols and functions are the same as in Explorer. On the other hand, Gist gives us an additional function *Gist node statistics*, which gives us more information about a specific node of the search tree, such as the depth of the node (according to a tree root), current height of the node's subtree and the number of successful/unsuccessful solutions in the subtree. In Gist there is also a tool Inspector, which works equally to the Inspector in Mozart/Oz.

4.4.3 Subjective description

Gecode is a typical example of a top software, which can be totally unusable due to the absence of a documentation. Before the version 3.0.0, which was released in March 2009, there was only a technical documentation generated by Doxygen. Furthermore, the example models were sealed into a pattern class Example to achieve a more simple call of the examples; however, it lacked any further explanations how to implement the solver inside own project and it was necessary to deeply examine the source code of the solver. Since the version 3.0.0 being released, there is available an e-book Modelling

with Gecode [20], which has completely changed the situation. The e-book is a tutorial, which step by step guides the reader from a basic query – from a given problem construct a model for the solver – to wide range of possibilities of setting up the solver. Apart from the classical documentation, there is also an e-mail conference, which is – while writing this thesis – quite a great place to ask and where the authors answer after a short period.

4.5 ECL^{i}PS^e

The ECL^{i}PS^e Constraint Programming System is an open source implementation of the Prolog programming language. ECL^{i}PS^e is provided with libraries for constraint solving. It is possible to solve models over integers, real numbers and finite integer sets. The solver is not just one library but there is one general solver and several specialized solvers. The basic solver is the *ic* library – interval constraints which contains basic arithmetic constraints. For global constraints there is an *ic_global* library. The global constraints are constraints which use a couple of more advanced techniques to filter the variable domain, for example the `alldifferent` constraint implemented as a matching in a bipartite graph. The ic library contains the alldifferent constraint too, but the one introduced in ic_global library is stronger. For scheduling problems there is an *ic_cumulative* library and for finite integer sets an *ic_sets* library. The users can define their own constraints if the shipped set of constraints is not satisfactory.

4.5.1 Solver description

As stated in the previous paragraph ECL^{i}PS^e is an implementation of the Prolog programming language. The system contains the solver as an independent library. Therefore the users are not limited to use only the shipped solvers, but they can use their own solvers if any. A standard solver is the *ic* solver which is a hybrid finite domain and real number interval constraint solver. As the name suggests, it offers constraining the variables with both real and integer domains. It supports arithmetic expressions, arithmetic constraints, global constraints, reified constraints and search algorithms. The set of global constraints can be extended by using *ic_global* constraints. This library provides constraints for lists such as `alldifferent`, `ordered`, `occurences` and so on. For scheduling the mentioned *ic_cumulative* constraint is available as well as their stronger versions *ic_edge_finder* or *ic_ed-*

ge_finder3, both of which vary in the time complexity of used algorithms. For the constraints over the symbolic domains – such as days of the week – the *ic_symbolic* domain is available. The constraining over finite integer domains is also available using the *fd_sets* library. This library provides member constraints, cardinality constraints, relation constraints, as well as set expressions. The problem is typically modeled as a Prolog predicate which constrains the variables domains, then applies constraints on these variables and, finally, it calls a search algorithm on the variables.

A capability of the solver can be extended by defining user defined constraints. For this purpose the system is equipped with *Propia* library and the *Constraint Handling Rules* library or *CHR*. CHR is a high level language for describing constraint rules. The reader can find the description of CHR in [19]. ECL^{i}PS^e offers *chr* library which can load a source code in CHR format, translate it into Prolog predicate and then include it. The constraint then can be used as any standard constraint shipped with the system. The second way to introduce new constraint is to use the Propia system. Propia takes any Prolog predicate and convert it to the proper constraint. The calling convention is `Goal infers most`. The library infers as much information about *Goal* as possible based on the loaded constraint solver libraries. The level of inference can be adjusted. Propia offers an approximate generalized propagation. The *most* inference can be expensive to compute and may not be necessary. The alternatives are predicates `Goal infers ic`, `Goal infers unique` and `Goal infers consistent`. As the name suggests, the *unique* infer ensures that all answers to the query are unique. The *consistent* inference can give answer if the query can be solved or not. If it can be solved, it additionally checks whether the constraint is already true or not. The *ic* inference is the same as the *most* inference except that the *most* is based on currently loaded solvers compared to *ic* which uses the specified solver. We shall show the example of using of our reimplemented $X\# > Y$ constraint which enforces that X is greater than Y using the standard $>$ operator. The problem predicate is the same as if we used only shipped constraints; however, the $>$ operator cannot handle constrained variables. And even if it could, it is not what we need. The standard behavior is to determine if X is greater than Y in the time of calling of the operator predicate and that is all. We need the operator to keep track on the variables and enforce the constraints as the domain updates due to the other constraint propagation. This is the point when the Propia comes. Once we mark the predicate as `infers most`, it is normal constraint

just like any other.

```
problem(X, Y) :-
  X :: [-100..100],
  Y :: [-100..100],
  (X > Y) infers most,  % predicate sent to Propia
  X #= 5,
  labeling([X,Y]).
```

4.5.2 Debugging support

ECL^{i}PS^e offers visualization related libraries. First the users have to create a viewable object using `create_viewable` from *viewable* library which contains variables in a proper order and so on. Then users have to invoke a visualization client, which is responsible for visualization of the given viewable objects. The visualization can be performed using a visualization client in the *java_vc* library. The created *viewable* objects are shown in the visualization client using several types of viewlets.

4.5.3 Subjective description

The ECL^{i}PS^e system has an exhaustive documentation [13] which covers both the Prolog language and the available libraries. The constraint library itself is more deeply documented in the constraint library manual [10]. The set of constraints which the solver offers is quite limited compared to the other solvers; however, the set of constraints can be extended either by using the external libraries or by implementing of the own constraints.

4.6 SICStus Prolog

SICStus Prolog is an implementation of the Prolog programming language by Swedish Institute of Computer Science or SICS. It differs from the other here discussed solvers because SICStus Prolog is not a free open source system. The trial version of this system can be obtained. Just like the ECL^{i}PS^e system the SICStus Prolog is not only a constraint solver but a programming language where constraint solver is shipped as an independent library. The system provides libraries for solving the constraint problems over the finite domains, over the boolean domains and over the real domains.

4.6.1 Solver description

Similarly as in the ECL^{i}PS^e system the solvers are independent libraries shipped with the system. The constraint solver over the finite domain consist of the *clp(fd)* library [12]. The language constructs are different compared to ECL^{i}PS^e ; however, the basic properties are preserved. Therefore porting the ECL^{i}PS^e program to SICStus Prolog is not a hard work but also it is not trivial. The number of available constraints is higher than the ECL^{i}PS^e system provides. The clp(fd)library offers standard arithmetic expressions, the relation constraints and the global constraints. The library also provides the scheduling constraints and the extensional constraints.

The *clp(b)* provides a solver over the boolean variables. The solver contains the relation constraints as well as constraints for the tautology and satisfiability. Finally the *clp(q,r)* library [16] provides constraining over the rational and real numbers. The constraint solver for set variables is not available.

The standard set of constraints can be expanded by the user defined constraints. As well as the ECL^{i}PS^e system the SICStus Prolog offers to add constraints using the Constraint Handling Rules language and FlatZinc language. User can also define his own constraints as the Prolog predicate. The user created constraint can be either a global constraint or a primitive constraint. The ways how to create such constraints are described in the SICStus Prolog manual.

The constraint library clfpd is very similar to the ECL^{i}PS^e library ic. We used the benchmarks already implemented in the ECL^{i}PS^e system and changed a very small amount of library-specific calls to get the SICStus Prolog source code.

4.6.2 Debugging support

The SICStus Prolog offers the *Finite Domain Constraint Debugger* or *fdbg* library. This library can be used next to the standard Prolog debugging predicate `trace`. This predicate allows to debug any Prolog program. The *fdbg* library provides the log of the constraint propagation and distribution. There are logged all changes to the domains and other events which occurred during the computation. The example of the *fdbg* output is in the figure 4.2. On the first line there is a Prolog query. We constrain the variable X to be in the domain $\{5, ..., 9\}$ and the variable Y to be in the domain $\{1, ..., 6\}$. Then we constrain the variable X to be smaller than variable

Figure 4.2: The example of the *fdbg* output.

```
| ?- X in 5..9, Y in 1..6, X #< Y.
<fdvar_9> in 5..9
    fdvar_9 = inf..sup -> 5..9
    Constraint exited.

<fdvar_10> in 1..6
    fdvar_10 = inf..sup -> 1..6
    Constraint exited.

<fdvar_9>+1#=<<fdvar_10>
    fdvar_9 = 5..9 -> {5}
    fdvar_10 = 1..6 -> {6}
    Constraint exited.

X in 4..6,
Y in 5..7
```

Y. We did not perform the search so the variables remains not solved. User can set names for the variables to achieve a better orientation in the printout. The library uses the following events which results in the action. The *constraint event* which is invoked when the global constraint is woken and the *labelling event* which is called after the variable labeling is started or a variable gets constrained or the labeling has failed. After such an event occurs the *visualizer* is called. The visualizer is a Prolog predicate which typically shows the event in the user friendly format; however, it can do any action instead as a response to the event. The primitive constraints are not tracked by *fdbg* but the arithmetical constraints like $X\# < Y$ are changed to the global constraints when the library is loaded. The user can specify the filename where the output of the *fdbg* will be written.

4.6.3 Subjective description

The SICStus Prolog is a professional solution. The variety of available libraries is really large; however if user need to solve constraint problem over the set variables it is useless. System is well documented and the manual is

exhaustive [4]. The system is not available freely for the public use but the time limited trial version can be acquired.

4.7 Tailor

Tailor is not a constraint solver. It is an open source Essence compiler which produces an input format of the Minion solver, a C++ source for the Gecode solver or an input format of Gecode/FlatZinc. Tailor is a Java application, which can be run either interactively, or as a command line utility. Tailor accepts as an input both an Essence program and an Essence parameters file. Then it normalizes the input into one source code. After normalization it performs the flattening of the code. During the flattening phase it replaces all the foreach loops and the sums with new auxiliary variables and with repeated basic constraints. Finally, it transforms this normalized code to a target language. Moreover, one can tune up the output of the Tailor solver by setting up several parameters about the language constructs which can be used. Another feature, which Tailor has, is that one can run the solver directly from its environment over the translated source code.

Chapter 5

Benchmark results

In this chapter we present the results of the performance measuring of each solver. There are two types of tests. The robustness test and the performance test. In the robustness test we compare the solvers according to the size of the task which the solver was able to compute in a given time limit. In the performance tests we measure the time which is needed to find a solution to the problem and the consumed memory. We used for this purpose designed tool. The tool runs the solver executable with given parameters. While the solver process is running the benchmarking tool periodically checks the `/proc/(pid)/stat` file and saves the current state. If the elapsed time is larger than the given limit the `SIGINT` signal is sent to the process. The process has one second to end and produce some output. If the process is still running one second after the `SIGINT` signal the `SIGKILL` signal is sent and process is killed. In the robustness test the tool tries to estimate the size of the task using the binary search.

All of the benchmarks were performed on a single dedicated computer Pentium IV, 3GHz (single core with hyper threading), 1GB RAM running the Debian Linux with Linux kernel 2.6.18. Except the Mozart/Oz solver and SICStus Prolog all solvers were compiled on the target machine. The precompiled binaries distribution was used for the Mozart/Oz solver because the source distribution contained in the time of writing of this thesis error which prevented compiling of the code. The SICStus Prolog is not distributed as source code but only as a binary. Therefore we used the shipped executable. Since the SICStus Prolog is not a free open-source software we used a trial version of the solver. There should be limited only the period when the solver can be used. The performance of the solver should not be

affected.

The following versions of the solvers were used:

- Mozart 1.4.0-20080704
- Choco 2.1.0
- Minion 0.8.1
- Gecode 3.1.0
- ECL^{i}PS^e 6.0_96
- SICStus Prolog 4.0.7 – Linux, glibc 2.7

In the performance test as well as in the robustness test we used the fist-fail search strategy for all of the implemented benchmarks. This strategy is the default strategy the all solvers.

5.1 The robustness test

In the robustness test we measure how long magic sequence the solver is able to compute in ten minutes. As stated in the previous paragraph the benchmarking tool uses for this purpose the binary search. The tool has an interval $[a, b]$ of values on which the search is performed. It picks a value v in the middle of the interval and tries to compute the solution of that length. If it success it changes the interval to $[v + 1, b]$, if it fails it changes the interval to $[a, v - 1]$. Then the search continues in the same way until the lower bound of the interval is larger than upper bound. The results of the robustness test are in the table 5.1.

The results of the robustness test are in the table 5.1 (the best result is bolded). As we can see the most robust solver is the Minion solver followed by the Gecode. At first we tried to implement the benchmark problem in the Gecode system by using of the constraint `count`. This constraint meaning is exactly the Magic Sequence problem; however, the longest magic sequence which we were able to compute had the length 27. If we compare this value with the current value 191 it is obvious that the implementation of this constraint is problematic.

Table 5.1: The results of the robustness test.

Mozart/Oz	Choco	Minion	Gecode	ECL^{i}PS^e	SICStus Prolog
94	111	**236**	191	55	174

5.2 The performance test

This test compares the solvers according to the two criteria. The first criterion is the time needed to solve given problem instances. The second criterion is the highest peak of used computer memory during the computation. We divided the instances of the benchmarks into two groups. In the first group there are the instances of the Quasigroup With Holes problem.

We generated 25 balanced quasigroups of the order $n = 16$ and removed approximately $1.6n^{1.55}$ values. According to the [5] the quasigroups with this number of the holes are hard to solve compared to the other quasigroup of the same order. To generate the quasigroups we used the *lsencode* combined with the *walksat*.

This benchmark gives us a comparison of the `allDifferent` constraint. This constraint is a global constraint. That means that the constraint is not binary but it constrain a set of variables. The allDifferent constraint is typically implemented using the matching in a bipartite graph [22]. The one group of nodes consists of the variables and the second group consists of the possible values of these variables. The variable node and the value node are connected by the edge if and only if the value is in the variables' domain.

As we can see in the graph in the figure 5.1 (a version with logarithmic scale is in the figure 5.2) the best results were achieved with Gecode, Mozart and Minion. The performance of the SICStus Prolog and ECL^{i}PS^e compared to the others is worse than the order of the magnitude.

In the second group of benchmarks there are the following instances of the benchmarks:

- *queens10* – find all solutions of the 10 queens problem,
- *queens100* –find one solution of the 100 queens problem,
- *magic20* – find the Magic sequence of the length 20,
- *srq* – find the solution of the Self Referential Quiz,

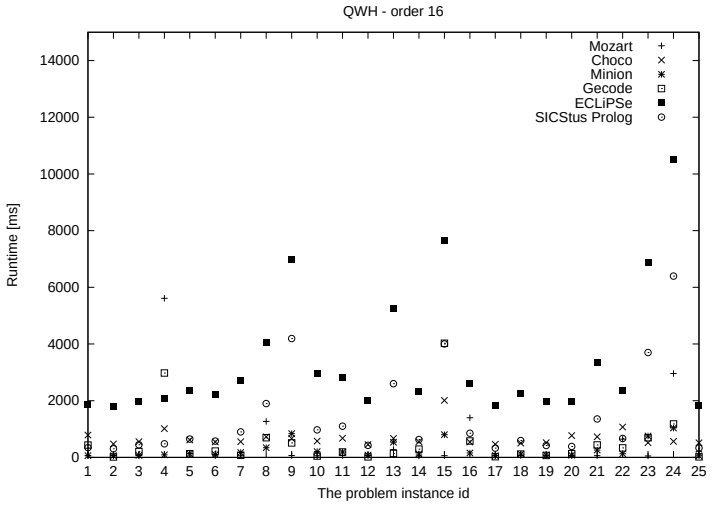


Figure 5.1: The QWH benchmark results

- *warehouses* – find the best solution of the Locating Warehouses problem.

As we can see in the tables 5.2 and 5.3 (the best result of each benchmark is bolded) none of the solvers is the ultimate winner; however, the three solvers – SICStus Prolog, Gecode and Mozart – achieved the best results in some of the problem instances. ECL^{i}PS^e and Minion did not solve the 100 queens problem in the given time limit. Therefore we consider them as unreliable. The ECL^{i}PS^e and Choco were unable to load the big problems to the memory so we changed the memory limit to one gigabyte. For the ECL^{i}PS^e we changed it for all of the experiments and for the Choco we used this large memory only for the magic sequence problem. The table 5.3 indicates that the Java Virtual Machine which is running the Choco as well as the ECL^{i}PS^e solver allocated the whole allowed amount of memory. We observer in the performance test the same problem with the `count` constraint in the Gecode system as in the robustness test. For a comparison the original results for the magic20 benchmark was 2926.86 milliseconds. The memory peak was nearly the same.

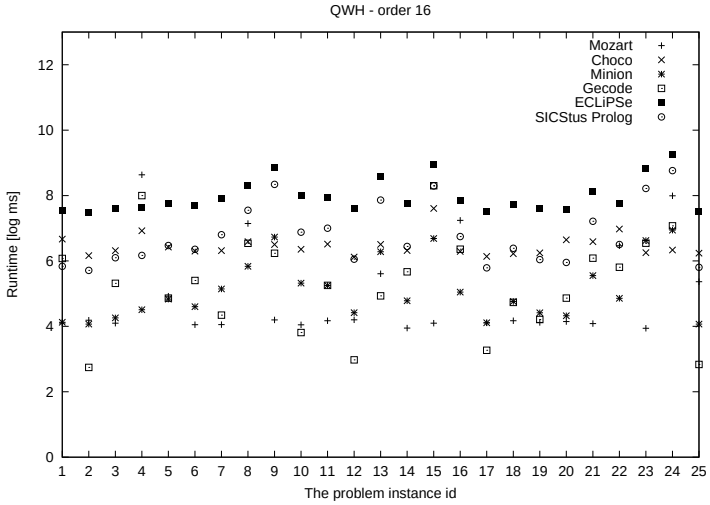


Figure 5.2: The QWH benchmark results, logarithmic scale

Table 5.2: The results of the performance test. Runtime (ms)

Benchmark	Moz	Cho	Min	Gec	ECL	SiP
queens10	237.95	850.27	227.69	**65.35**	809.45	327.53
queens100	184.19	1005.02	×	34.70	×	**15.40**
magic20	**11.48**	1092.95	71.24	65.41	4514.14	198.32
srq	**6.22**	482.81	55.81	25.89	691.13	13.54
warehouses	68.28	1500.83	2338.45	169.71	1808.29	**14.41**

Table 5.3: The results of the performance test. Memory peak (MB)

Benchmark	Moz	Cho	Min	Gec	ECL	SiP
queens10	8.62	209.42	44.17	**8.58**	1158.15	38.08
queens100	23.31	209.46	×	6.71	×	**1.35**
magic20	**2.32**	1173.19	44.57	7.26	1157.99	39.30
srq	63.36	209.35	44.56	6.68	1158.08	**0.94**
warehouses	6.79	209.12	44.30	7.00	1157.99	**0.89**

Chapter 6

Conclusions

This thesis presented a survey of available constraint solvers and more detailed described six of them. After reading of this thesis the reader should have a basic idea about capabilities and performance of each of them. The solver capabilities were compared on the benchmark problems which were implemented in all of them. On these benchmarks we showed a different approach of each solver how to represent the constraint problem model.

6.1 Which solver to choose?

It is a hard question and there is no simple answer to it. Each solver has an area where is the best choice of others. If we are forced to use a particular language the best choice would be the solver in that language. That means that for C++ projects we choose Gecode, for Java projects we use Choco, for Oz projects the Mozart/Oz and finally for Prolog project we can choose the ECL^{i}PS^e or SICStus Prolog. Since all the solvers except SICStus Prolog are free we can choose any of them if the main criterion is the budget. If we need to model the exotic constraints we cannot use the Minion system. On the other hand the Minion system is suitable as an external solver for environments where is easy to generate an input file, run a process and then process its output. Example of such environment could be the bash script language. Last but not least criterion can be the readers' conservatism and will to learn new things. For example the Mozart/Oz is an interesting solver and language but if the reader do not want to get familiar with the functional programming paradigm that solver would not be suitable for him.

Based on the experimental results the Choco and ECL^{i}PS^e can be labeled

as "slow systems". ECL^{i}PS^e has also the limited set of constraint compared to the other solvers. The set can be extended by the user but the other system offers wider set of the constraints in the basic release. Minion has a limited set of constraints which cannot be extended. Moreover it failed to compute one of the benchmarks. Mozart/Oz is quite fast system; however the set of available constraints is also limited and the system development seems stopped.

This leaves us the SICStus Prolog and Gecode which can be recommended as the first choice. The SICStus Prolog was the best in most of the performance benchmarks and offers many specialized constraints; however, it is not a free open source system and the price has to be considered. Also the results of the Quasigroup With Holes benchmark was one the worst compared to the other solvers. Gecode offers nearly as large set of constraints as SICStus Prolog. Moreover Gecode offers constraining over the set variables which is not available in the SICStus Prolog.

6.2 Future work

We did not perform a deeper study of the search strategies of the solvers. In the future it would be useful to compare the other strategies which are shipped with the solver. In the solvers which offers ways to implement the own strategy we should try to implement such a strategy and compare it with the shipped strategies. Also the comparison of the solver based on the Quasigroup With Holes problem can be performed on the quasigroups of the higher order (at least 30). Such a comparison should give us a more accurate data about the performance of each solver. Finally we could examine the solvers on the real world applications for example as a solver subsystem of a larger software.

Bibliography

[1] *Mozart Documentation*, 2008. [online, available at http://www.mozart-oz.org/documentation/; accessed 06-August-2009].

[2] *Choco solver, Documentation*, 2009. [online, available at http://www.emn.fr/x-info/choco-solver/doku.php?id=documentation; accessed 06-August-2009].

[3] *Gecode Reference Documentation 3.1.0*, 2009. [online, available at http://www.gecode.org/doc-latest/reference/index.html; accessed 06-August-2009].

[4] *SICStus Prolog User's Manual, release 4.0.7*, 2009. [online, available at http://www.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf; accessed 06-August-2009].

[5] Dimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. Generating satisfiable problem instances. In *AAAI/IAAI*, pages 256–261. AAAI Press, 2000.

[6] The National Aeronautics and Space Administration. Deep space 1: Autonomous remote agent. [online, available at http://nmp.nasa.gov/ds1/tech/autora.html; accessed 02-December-2008], 2001.

[7] Roman Barták. Omezující podmínky: od sudoku po vesmírné aplikace. In *Umělá inteligence (5)*, pages 146–172. Academia, 2007. In czech language.

[8] Roman Barták. On-line guide to constraint programming – systems, 2007. [online, available at http://ktiml.mff.cuni.cz/ bartak/constraints/systems.html; accessed 21-July-2008].

[9] Roman Barták. On-line guide to constraint programming, 2009. [online, available at http://ktiml.mff.cuni.cz/ bartak/constraints/index.html; accessed 23-July-2008].

[10] Pascal Brisset et al. *ECL^{i}PS^e, Constraint Library Manual*, 2009. [online, available at http://87.230.22.228/doc/libman.pdf; accessed 06-August-2009].

[11] Maja Bubalo and Mirko Čubrilo. Modeling and solving self-referential puzzles. *JIOS*, 29(1):268–306, 2005.

[12] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. Technical report, Proc. Programming Languages: Implementations, Logics, and Programs, 1997.

[13] Andrew M. Cheadle et al. *ECL^{i}PS^e, A Tutorial Introduction*, 2009. [online, available at http://87.230.22.228/doc/tutorial.pdf; accessed 06-August-2009].

[14] Antonio J. Fernández and Patricia M. Hill. A comparative study of eight constraint programming languages over the boolean and finite domains. *Constraints*, 2002.

[15] Ian Gent and Toby Walsh. Csplib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[16] Christian Holzbaur. Ofai clp(q,r) manual. Technical report, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[17] Christopher Jefferson et al. *The Minion Manual, Minion Version 0.8.1*, 2009. [online, available at http://minion.sourceforge.net/files/Manual081.pdf; accessed 06-August-2009].

[18] Miroslava Plachá. Porovnání systémů pro programování s omezujícími podmínkami. Bachelor thesis, Fakulta Informatiky, Masarykova univerzita, Brno, 2007. In czech language.

[19] Tom Schrijvers. Constraint handling rules (CHR). [online, available at http://www.cs.kuleuven.be/ dtai/projects/CHR/; accessed 26-July-2009], 2009.

[20] Christian Schulte, Guido Tack, and Mikael Zayenz Lagerkvist. *Modeling with Gecode*, 2009. [online, available at http://www.gecode.org/doc-latest/modeling.pdf; accessed 06-August-2009].

[21] Marc van Dongen. Call for solvers and benchmarks fourth international constraint solver competition (csp, max-csp and weighted-csp competition). [online, available at http://cpai.ucc.ie/09/call2009.pdf; accessed 26-July-2009], 2009.

[22] Willem-Jan van Hoeve. The alldifferent constraint: A survey. Sixth Annual Workshop of the ERCIM Working Group on Constraints, Prague, 2001.

Appendix A

Contents of the CD

We include a CD with additional content to the thesis. On the included CD one can find the installation files of the discussed solvers except SICStus Prolog. Also we put there full source code of the benchmarks implemented in all solvers including minion input files. An electronic copy of this thesis can be found there in the PDF format.

The content of the CD is available at http://www.tulacek.eu/bc_thesis/

A.1 Files and directories

- *readme.txt* – the detailed contents of the CD
- *thesis.pdf* – the electronic version of this thesis
- *systems* – a directory with the installation files
- *benchmarks* – a directory with the source codes of benchmarks
  - *data* – the raw test results and the QWH assignments
  - *choco* – the implementation in the Choco
  - *eclipse* – the implementation in the ECL^{i}PS^e
  - *gecode* – the implementation in the Gecode
  - *minion* – the implementation in the Minion
  - *mozart* – the implementation in the Mozart/Oz
  - *sicstus* – the implementation in the SICStus Prolog

Appendix B

List of the constraint solvers

We present here a short list of the constraint solvers. The more detailed list can be found in [8].

CHIP V5

CHIP (constraint handling in Prolog) is the second-generation constraint programming tool. It supports modeling of the scheduling, logistics and manpower planning.

URL: http://www.cosytec.com/production_scheduling/chip/optimization_product_chip.htm

Choco

Choco is a solver library written in the Java programming language.

URL: http://www.emn.fr/x-info/choco-solver/

Comet

Comet is an Object Oriented Programming Language for Constraint-Based Local Search.

URL: http://www.comet-online.org/

ECL^{i}PS^e

ECL^{i}PS^e is an implementation of the Prolog programing language focused on the constraint programming.

URL: http://www.eclipse-clp.org/

Gecode

Gecode is a solver library written in the C++ library.

URL: http://www.gecode.org/

Gecode/FlatZinc

Gecode/FlatZinc is a solver for the problems modeled in the FlatZinc language

URL: http://www.gecode.org/flatzinc.html

Gecode/J

Gecode/J is the implementation of the Gecode solver in the Java programming language. The software is no longer actively developed.

URL: http://www.gecode.org/gecodej/

Gecode/R

Gecode/R is the implementation of the Gecode solver in the Ruby programming language.

URL: http://gecoder.rubyforge.org/

ILOG OPL Development Studio

ILOG OPL provides a set of tools to model and solve the constraint problems. The models can be integrated into Java, C++ or .NET applications.

URL: http://www.ilog.com/products/oplstudio/

Minion

Minion is a standalone solver which uses its own low-level language to describe the constraint model.

URL: http://minion.sourceforge.net/

Mozart/Oz

A multiparadigmatic language with the built-in support for constraint solving.

URL: http://www.mozart-oz.org/

SICStus Prolog

SICStus Prolog is an implementation of the Prolog programming language with the finite domain library.

URL: http://www.sics.se/isl/sicstuswww/site/

Sugar

Sugar is a SAT-based constraint solver. Sugar has won the *global constraints* category in the CSP 2008 Competition.

URL: http://bach.istc.kobe-u.ac.jp/sugar/

Appendix C

List of the constraints

In this appendix we present the comparison of modelling capabilites of the solvers based on the constraints which are supported. We used the manuals of the systems [1–4, 10, 13, 17, 20] as the source of the data. We do not want to provide a full list of the supported constraints. Our goal is to show the most important constraints or the constraints which defines the differences between the solvers. For each group of the constraint we present a comparison table. The used symbols have the following meaning: The constraint variables are denoted by the emphatised small latin letters (x, y, a). The constants are symbolised by the greek small letters (α, β, γ). The vectors of the constraint variables or the constants are in bold ($\mathbf{x}$, $\mathbf{a}$, $\boldsymbol{\alpha}$). As usual in programming languages by square brackets applied on the vector we denote the element of a vector ($\mathbf{x}[y]$). Finally the boolean variables are denoted by the monospace font ($\mathtt{x}, \mathtt{b}$). The symbol tilde ($\sim$) means a relation operator (any of the $=$, $\neq$, $>$, $<$, $\leq$, $\geq$). The other symbols have their usual meaning. If it is not possible to describe the constraint with the mathematical expression we use a symbolic name instead. If the solver fully supports the constraint the $\bullet$ symbol is used. If the constraint is supported only partial the symbol $\circ$ is used. If the solver does not support the constraint at all we use the symbol $\times$. We say that the solver supports a constraint only if the constraints is available as a single constraint which can be directly used or it there exists an expression similar to the constraint. In the other case even if the constraint effect can be achieved by the combination of some other constraints we consider that constraint as unsupported. We use the following abbreviations for the constraint solvers: Moz means Mozart/Oz, Cho means Choco, Min means Minion, ECL means ECL^{i}PS^e and finally SiP

Table C.1: Relation and arithmetic constraints

Constraint	Moz	Cho	Min	Gec	ECL	SiP
$x \sim y$	●	●	○	●	●	●
$\sum_i \mathbf{x}_i \sim y$	●	●	○	●	●	●
$\sum_i (\mathbf{x}_i \mathbf{w}_i) \sim y$	●	●	○	●	×	●
$\sum_i \left(\mathbf{x}_i \prod_j \mathbf{w}_{ij} \right) \sim y$	●	×	×	×	×	×
$\bigwedge_i \mathtt{x}_i \wedge \bigwedge_i \neg \mathtt{y}_i = \mathtt{b}$	×	×	×	●	×	×
$\bigvee_i \mathtt{x}_i \vee \bigvee_i \neg \mathtt{y}_i = \mathtt{b}$	×	×	×	●	×	×
$\lvert x \rvert = y$	●	●	●	●	●	●
$\lvert x - y \rvert \sim z$	●	×	○	×	×	●
$x \bmod y = z$	●	●	●	●	●	●

means SICStus Prolog.

We divide the constraints into four groups – *relation and arithmetic*, *global*, *scheduling* and *other* constraints. The most of the relation and arithmetic constraints (see the table C.1) are supported in the all solvers. As we can see the Minion system has for some of the constraints only the partial support. For the sums it means that Minion supports only the $\leq$ and $\geq$ operators. The partialy supported constraint $x \sim y$ on the other hand means that it supports only the $=$ and $\neq$ operators and finally for the $|x - y| \sim z$ constraint only the operator $=$ is supported. The global constraints (see the table C.2) are the constraints which constrain the vectors of variables. The basic global constraint is *alldifferent* which constrains the given vector $\mathbf{x}$ such that each variable contained in the vector has a different value. The *hamming* constraints ensures that the hamming distance of the two given vectors is at least c. The scheduling constraints as the name suggest offers the support for modelling of the scheduling problems. The *cummulative* constraint is used if we need to schedule the tasks on the given machines. The *overlaps* and the *disjunctive* are used to constrain the relative order of the taks. Finally in the fourth group there are the other constraints. By using the *extensional* constraint we can constrain the complex relations of the variables by specifying the set of compatible values. The constraint then constrain the variables such that they have only the compatible values. The DFA stands for the deterministic fiinite automaton. The constraint enforces that the given vector of variables is the word which is accepted by the solver.

Table C.2: Global constraints

Constraint	Moz	Cho	Min	Gec	ECL	SiP
alldifferent($\mathbf{x}$)	•	•	•	•	•	•
$\mathbf{x}[i] = j \Leftrightarrow \mathbf{y}[j] = i$	×	•	×	•	×	•
hamming($\mathbf{x}, \mathbf{y}$) $\geq c$	×	×	•	×	×	×
$\mathbf{x}[y] = z$	•	•	•	•	•	•

Table C.3: Scheduling constraints

Constraint	Moz	Cho	Min	Gec	ECL	SiP
cummulative	×	•	×	•	•	•
overlaps	•	×	×	×	×	×
disjunctive	•	•	×	×	•	•

Finally the hamiltonian circle ensures that the given vector of variables is a hamiltonian circle in the graph. In the ECL^{i}PS^e we can define the extensional constraint by using of the Propia library. We define the predicate with the compatible values and the use the Propia library tools on that predicate.

Table C.4: Other constraints

Constraint	Moz	Cho	Min	Gec	ECL	SiP
extensional	×	•	•	•	○	•
DFA	×	•	×	•	×	•
hamiltonian circle	×	×	×	•	×	•

Appendix D

Implementation of the benchmarks

D.1 Mozart/Oz

The following code shows the implementation of the Locating Warehouses problem in the Mozart/Oz solver.

```
1  fun {Warehouses Data}
2
3     fun {Regret X}
4        M = {FD.reflect.min X}
5     in
6        {FD.reflect.nextLarger X M} - M
7     end
8
9     NbStores = {Width Data.costs}
10    NbWarehouses = {Width Data.capacity}
11    Capacity = Data.capacity
12    WarehouseCost = Data.warehouseCost
13    CostMatrix = Data.costs
14
15 in
16    proc {$ Sol}
17       TotalCost = {FD.decl}
18       Open = {FD.tuple warehouse 5 0#1}
19       Supplier = {FD.tuple supplier NbStores 1#NbWarehouses}
20
21       Cost = {FD.tuple store NbStores 0#FD.sup}
22       SumCost = {FD.decl} = {FD.sum Cost '=:'}
23
24       Stores = {List.number 1 NbStores 1}
25
26       NbOpen = {FD.decl} = {FD.sum Open '=:'}
27    in
28       Sol = plan(totalCost:TotalCost open:Open supplier:Supplier)
29
30       TotalCost =: SumCost + NbOpen * WarehouseCost
31
32       {For 1 NbStores 1
33        proc {$ Store}
34     Cost.Store :: {Record.toList CostMatrix.Store}
35     {FD.element Supplier.Store CostMatrix.Store Cost.Store}
36        end}
37
38       {For 1 NbWarehouses 1
39        proc {$ Warehouse}
```

```
40      {FD.atMost Capacity.Warehouse Supplier Warehouse}
41      Open.Warehouse = {FD.reified.sum {Map Stores fun {$ Store} Supplier.Store =:
            Warehouse end} '>:' 0}
42         end}
43
44        {FD.distribute ff Supplier}
45     end
46 end
47
48 proc {Order Old New}
49     Old.totalCost >: New.totalCost
50 end
```

D.2 Choco

The following code shows the implementation of the self referential quiz in the Choco solver.

```
1  Model m = new CPModel();
2  Solver s = new CPSolver();
3
4  IntegerVariable[][] questions = new IntegerVariable[10][];
5
6  for (int i = 0; i < 10; i++)
7  {
8    questions[i] = makeBooleanVarArray("questions_"+i, 5);
9
10   m.addConstraint( eq( sum(questions[i]), 1));
11
12   m.addVariables(questions[i]);
13 }
14
15 // Question 1 A-D
16 for (int i = 0; i < 4; i++)
17 {
18   Constraint c = eq(questions[4-i-1][0], 1);
19
20   for (int j = 0; j < 4-i-1; j++)
21   {
22     c = and( c, eq(questions[j][0], 0) );
23   }
24
25   m.addConstraint( ifOnlyIf( eq(questions[0][i], 1), c ) );
26 }
27
28 // Question 1 E
29 Constraint c1e = TRUE;
30 for (int i = 0; i < 4; i++)
31 {
32   c1e = and( c1e, eq( questions[i][0], 0 ) );
33 }
34 m.addConstraint( ifOnlyIf( eq(questions[0][4], 1), c1e ) );
35
36 // Question 2
37 for (int i = 0; i < 5; i++)
38 {
39   Constraint c = TRUE;
40
41   for (int j = 0; j < 5; j++)
42   {
43     c = and( c, eq( questions[2+i][j], questions[3+i][j] ) );
44   }
45
46   m.addConstraint( ifOnlyIf( eq(questions[1][i], 1), c ) );
47 }
48
49 // Question 3
50 for (int i = 0; i < 5; i++)
51 {
52   Constraint c = eq( questions[i+3][0] , 1 );
53
```

```
54    for (int j = 0; j < i; j++)
55    {
56      c = and( c, eq( questions[j+3][0], 0 ) );
57    }
58
59    m.addConstraint( ifOnlyIf( eq(questions[2][i], 1), c ) );
60  }
61
62  // Question 4
63  for (int i = 0; i < 5; i++)
64  {
65    Constraint c = eq( questions[i*2+1][1] , 1 );
66
67    for (int j = 0; j < i; j++)
68    {
69      c = and( c, eq( questions[j*2+1][1], 0 ) );
70    }
71
72    m.addConstraint( ifOnlyIf( eq(questions[3][i], 1), c ) );
73  }
74
75  // Question 5
76  for (int i = 0; i < 5; i++)
77  {
78    Constraint c = eq( questions[i*2][2] , 1 );
79
80    for (int j = 0; j < 5; j++)
81    {
82      if (j != i)
83      {
84        c = and( c, eq( questions[j*2][2], 0 ) );
85      }
86    }
87
88    m.addConstraint( ifOnlyIf( eq(questions[4][i], 1), c ) );
89  }
90
91  // Question 6
92  IntegerVariable sumBefore = makeIntVar("question_6_sumBefore", 0, 10);
93  IntegerVariable sumAfter  = makeIntVar("question_6_sumAfter", 0, 10);
94  IntegerVariable[] before = new IntegerVariable[5];
95  IntegerVariable[] after = new IntegerVariable[4];
96
97  for (int i = 0; i < 5; i++)
98  {
99    before[i] = questions[i][3];
100 }
101
102 for (int i = 0; i < 4; i++)
103 {
104   after[i] = questions[6+i][3];
105 }
106
107 m.addConstraint( eq(sum(before), sumBefore) );
108 m.addConstraint( eq(sum(after), sumAfter) );
109
110 m.addConstraint( ifOnlyIf( eq(questions[5][0], 1), and( gt(sumBefore, 0), eq(sumAfter,
        0) )) );
111 m.addConstraint( ifOnlyIf( eq(questions[5][1], 1), and( eq(sumBefore, 0), gt(sumAfter,
        0) )) );
112 m.addConstraint( ifOnlyIf( eq(questions[5][2], 1), and( gt(sumBefore, 0), gt(sumAfter,
        0) )) );
113 m.addConstraint( ifOnlyIf( eq(questions[5][3], 1), and( and(eq(questions[5][3], 0), eq(
        sumBefore, 0)), eq(sumAfter, 0) )) );
114 m.addConstraint( ifOnlyIf( eq(questions[5][4], 1), eq(questions[5][3], 1)) );
115
116 // Question 7
117 for (int i = 0; i < 5; i++)
118 {
119   Constraint c = eq( questions[i+4][4] , 1 );
120
121   for (int j = i+4+1; j < 10; j++)
122   {
123     c = and( c, eq( questions[j][4], 0 ) );
124   }
125
126   m.addConstraint( ifOnlyIf( eq(questions[6][i], 1), c ) );
127 }
```

```
128
129
130 // Question 8
131 IntegerVariable sumConsonants = makeIntVar("question_8_sumConsonants", 0, 10);
132 IntegerVariable[] consonants = new IntegerVariable[3*10];
133 for (int i = 0; i < 10; i++)
134 {
135   consonants[3*i]     = questions[i][1];
136   consonants[3*i + 1] = questions[i][2];
137   consonants[3*i + 2] = questions[i][3];
138 }
139 m.addConstraint( eq(sum(consonants), sumConsonants) );
140
141 for (int i = 0; i < 5; i++)
142 {
143   m.addConstraint( ifOnlyIf( eq(questions[7][i], 1), eq(sumConsonants, 7 - i) ) );
144 }
145
146
147 // Question 9
148 IntegerVariable sumVowels = makeIntVar("question_9_sumVowels", 0, 10);
149 IntegerVariable[] vowels = new IntegerVariable[2*10];
150 for (int i = 0; i < 10; i++)
151 {
152   vowels[2*i]     = questions[i][0];
153   vowels[2*i + 1] = questions[i][4];
154 }
155 m.addConstraint( eq(sum(vowels), sumVowels) );
156
157 for (int i = 0; i < 5; i++)
158 {
159   m.addConstraint( ifOnlyIf( eq(questions[8][i], 1), eq(sumVowels, i) ) );
160 }
161
162 s.read(m);
163
164 Boolean some = s.solve();
165
166 if (some)
167 {
168     do {
169         for (int i = 0; i < questions.length; i++) {
170
171           for (int j = 0; j < questions[i].length; j++)
172             {
173             int value = s.getVar(questions[i][j]).getVal();
174               System.out.print(value+" ");
175             }
176             System.out.println();
177         }
178     } while (s.nextSolution());
179 }
```

D.3 Minion

The following code shows the Minion input file for the Magic Sequence problem of the length 5.

```
1  MINION 3
2
3  **VARIABLES**
4  DISCRETE s[5] {0..5}
5
6  BOOL aux0
7  BOOL aux1
8  BOOL aux2
9  BOOL aux3
10 BOOL aux4
11 BOOL aux5
12 BOOL aux6
13 BOOL aux7
```

```
BOOL aux8
BOOL aux9
BOOL aux10
BOOL aux11
BOOL aux12
BOOL aux13
BOOL aux14
BOOL aux15
BOOL aux16
BOOL aux17
BOOL aux18
BOOL aux19
BOOL aux20
BOOL aux21
BOOL aux22
BOOL aux23
BOOL aux24

**SEARCH**

PRINT[s]

VARORDER [
s , aux0, aux1, aux2, aux3, aux4, aux5, aux6, aux7, aux8, aux9, aux10, aux11, aux12,
    aux13, aux14, aux15, aux16, aux17, aux18, aux19, aux20, aux21, aux22, aux23, aux24]


**CONSTRAINTS**

reify(eq(0, s[0]), aux0)
reify(eq(0, s[1]), aux1)
reify(eq(0, s[2]), aux2)
reify(eq(0, s[3]), aux3)
reify(eq(0, s[4]), aux4)
sumleq([aux0,aux1,aux2,aux3,aux4], s[0])
sumgeq([aux0,aux1,aux2,aux3,aux4], s[0])
reify(eq(1, s[0]), aux5)
reify(eq(1, s[1]), aux6)
reify(eq(1, s[2]), aux7)
reify(eq(1, s[3]), aux8)
reify(eq(1, s[4]), aux9)
sumleq([aux5,aux6,aux7,aux8,aux9], s[1])
sumgeq([aux5,aux6,aux7,aux8,aux9], s[1])
reify(eq(2, s[0]), aux10)
reify(eq(2, s[1]), aux11)
reify(eq(2, s[2]), aux12)
reify(eq(2, s[3]), aux13)
reify(eq(2, s[4]), aux14)
sumleq([aux10,aux11,aux12,aux13,aux14], s[2])
sumgeq([aux10,aux11,aux12,aux13,aux14], s[2])
reify(eq(3, s[0]), aux15)
reify(eq(3, s[1]), aux16)
reify(eq(3, s[2]), aux17)
reify(eq(3, s[3]), aux18)
reify(eq(3, s[4]), aux19)
sumleq([aux15,aux16,aux17,aux18,aux19], s[3])
sumgeq([aux15,aux16,aux17,aux18,aux19], s[3])
reify(eq(4, s[0]), aux20)
reify(eq(4, s[1]), aux21)
reify(eq(4, s[2]), aux22)
reify(eq(4, s[3]), aux23)
reify(eq(4, s[4]), aux24)
sumleq([aux20,aux21,aux22,aux23,aux24], s[4])
sumgeq([aux20,aux21,aux22,aux23,aux24], s[4])

**EOF**
```

D.4 Gecode

The following code shows the implementation of the Locating Warehouses problem in the Gecode solver. The code is complete except the loading of

the data from the external source.

```
#include <gecode/int.hh>
#include <gecode/search.hh>
#include <gecode/minimodel.hh>

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace Gecode;

class Warehouses : public MinimizeSpace {
protected:
  IntVar TotalCost;
  IntVar NumberOpen;

  IntVarArray Open;
  IntVarArray Supplier;


  IntVarArray Cost;
  IntVar SumCost;

  int numberOfWarehouses;
  int numberOfStores;

public:
  Warehouses(
    int nbWarehouses,
    int nbStores,
    const std::vector<int>& supplyCost,
    const std::vector<int>& warehouseCapacity
    )
  :
    numberOfWarehouses(nbWarehouses),
    numberOfStores(nbStores),
    Supplier(*this, nbStores, 0, nbWarehouses-1),
    TotalCost(*this, 0, Gecode::Int::Limits::max),
    Open(*this, nbWarehouses, 0, 1),
    NumberOpen(*this, 0, nbWarehouses),
    Cost(*this, nbStores, 0, Gecode::Int::Limits::max),
    SumCost(*this, 0, Gecode::Int::Limits::max)
  {

    for (int i = 0; i < numberOfWarehouses; i++)
    {
      IntVarArray aux_supplied(*this, numberOfStores, 0, 1);

      for (int j = 0; j < numberOfStores; j++)
      {
        post(*this, tt( imp( Supplier[j] == i, aux_supplied[j] == 1) ));
        post(*this, tt( imp( Supplier[j] != i, aux_supplied[j] == 0) ));

        post(*this, tt( imp( Supplier[j] == i, Cost[j] == supplyCost[i + j *
            numberOfWarehouses]) ));
      }

      IntVar aux_sum(*this, 0, numberOfStores);

      linear(*this, aux_supplied, IRT_EQ, aux_sum);

      post(*this, tt( imp( aux_sum > 0, Open[i] == 1 ) ));
      post(*this, tt( imp( aux_sum == 0, Open[i] == 0 ) ));

      rel( *this, aux_sum, IRT_LQ, warehouseCapacity[i] );
    }

    linear(*this, Open, IRT_EQ, NumberOpen);

    linear(*this, Cost, IRT_EQ, SumCost);

    IntArgs costArgs(2);
    costArgs[0] = 1;
    costArgs[1] = 50;

    IntVarArgs costVariables(2);
    costVariables[0] = SumCost;
```

```
77      costVariables[1] = NumberOpen;
78
79      linear(*this, costArgs, costVariables, IRT_EQ, TotalCost);
80
81      branch(*this, Supplier, INT_VAR_SIZE_MIN, INT_VAL_MIN);
82    }
83
84    virtual IntVar cost() const {
85      return TotalCost;
86    }
87
88    Warehouses(bool share, Warehouses& s) : MinimizeSpace(share, s) {
89      Supplier.update(*this, share, s.Supplier);
90      TotalCost.update(*this, share, s.TotalCost);
91      Open.update(*this, share, s.Open);
92      NumberOpen.update(*this, share, s.NumberOpen);
93      Cost.update(*this, share, s.Cost);
94      SumCost.update(*this, share, s.SumCost);
95
96      numberOfWarehouses = s.numberOfWarehouses;
97      numberOfStores = s.numberOfStores;
98    }
99
100   virtual Space* copy(bool share)
101   {
102     return new Warehouses(share, *this);
103   }
104
105   void print(void) const {
106     std::cout << "Cost: " << TotalCost << std::endl;
107     std::cout << "Suppliers:" << Supplier << std::endl;
108     std::cout << "Open:" << Open << std::endl;
109   }
110
111 };
112
113 int main(int argc, char** argv)
114 {
115   std::vector<int> supplyCost;
116   std::vector<int> warehouseCapacity;
117
118   // There should be the loading of the data into the vectors.
119   // We left it out in printout
120
121   Warehouses* w = new Warehouses( warehouseCapacity.size(), supplyCost.size()/
          warehouseCapacity.size(), supplyCost, warehouseCapacity);
122
123   BAB<Warehouses> e(w);
124   delete w;
125
126   Warehouses* last = 0;
127   while (Warehouses* s = e.next())
128   {
129     if (last != 0)
130     {
131       delete last;
132     }
133
134     last = s;
135   }
136
137   if (last)
138   {
139     last->print();
140     delete last;
141   }
142
143   return 0;
144 }
```

D.5 ECL^{i}PS^e

The following code shows the implementation of the N-Queens problem in the ECL^{i}PS^e solver.

```
:- lib(ic).

all_queens(N, Out) :-
  bagof(Sol, queens(N, Sol), Out).

queens(N, Sol) :-
  length(Queens, N),
  Queens :: [1..N],
  alldifferent(Queens),
  diagonals(Queens),
  labeling(Queens),
  Sol = Queens.

diagonals([H|T]) :- diagonals([H|T], T).

diagonals([], _).
diagonals(_, []).
diagonals( [H1|T1], [H2|T2]) :- diagonals_mark(H1, [H2|T2], 1), diagonals(T1, T2).

diagonals_mark(_, [], _).
diagonals_mark(H, [H2|T2], A) :-
  H - H2 #\= A,
  H - H2 #\= -A,
  AA is A + 1,
  diagonals_mark(H, T2, AA).
```

D.6 SICStus Prolog

The following code shows the implementation of the Quasigroup With Holes in the SICStus Prolog.

```
:- use_module(library(clpfd)).

qwh(N, Assignment, Sol) :-
  NN is N*N,
  length(Matrix, NN),
  domain(Matrix, 1, N),
  q_constraint(N, Matrix),
  assign_constraint(Matrix, Assignment),
  labeling([], Matrix),
  Sol = Matrix.

write_solution(N, Sol) :-
  NN is N - 1, write_solution(N, NN, Sol, 0).

assign_constraint([], []).
assign_constraint([_|T], [-1|TT])
  :- !, assign_constraint(T, TT).

assign_constraint([H|T], [HH|TT]) :-
  H #= HH,
  !,
  assign_constraint(T, TT).

write_solution(_, _, [], _).
write_solution(N, NN, [H|T], A) :-
  M is A mod N,
  write(H),
  (M = NN, write('\n') ; M \= NN, write(' ')),
  AA is A + 1,
  write_solution(N, NN, T, AA).

select_column(N, ColId, Matrix, Col) :-
```

```
33    select_column(N, ColId, Matrix, Col, 0).
34
35  select_column(_, _, [], [], _).
36  select_column(N, ColId, [H|T], [H|CT], A) :-
37    ColId is A mod N,
38    AA is A + 1,
39    select_column(N, ColId, T, CT, AA).
40  select_column(N, ColId, [_|T], CT, A) :-
41    Mod is A mod N,
42    Mod \= ColId,
43    AA is A + 1,
44    select_column(N, ColId, T, CT, AA).
45
46
47  select_row(N, RowId, Matrix, Row) :-
48    BoundL is RowId*N,
49    BoundH is (RowId+1)*N-1,
50    select_row(N, Matrix, Row, 0, BoundL, BoundH).
51
52  select_row(_, [], [], _, _, _) :- !.
53  select_row(_, _, [], A, _, BH) :- BH < A, !.
54
55  select_row(N, [H|T], [H|RT], A, BL, BH) :-
56    A >= BL,
57    A =< BH,
58    AA is A + 1,
59    !,
60    select_row(N, T, RT, AA, BL, BH).
61
62  select_row(N, [_|T], RT, A, BL, BH) :-
63    A < BL,
64    AA is A + 1,
65    !,
66    select_row(N, T, RT, AA, BL, BH).
67
68
69  q_constraint(N, Matrix) :-
70    q_constraint(N, Matrix, 0).
71
72  q_constraint(N, _, M) :- M >= N, !.
73  q_constraint(N, Matrix, A) :-
74    A < N,
75    select_row(N, A, Matrix, Row),
76    select_column(N, A, Matrix, Col),
77    all_different(Row),
78    all_different(Col),
79    AA is A + 1,
80    !,
81    q_constraint(N, Matrix, AA).
```