A Modeling Language for Constraint Programming

ANDREA RENDL

Examiner
Dr. Christian Schulte

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-58

A Modeling Language for Constraint Programming

Andrea Rendl

March 2006

Master's Thesis

Supervisor: Dr. Christian Schulte, Associate Professor, KTH ICT Stockholm
Univ.Prof. Dipl.-Ing. Dr. Wolfgang Slany

This Thesis

This is a Master's Thesis from the Telematics Master programme at the University of Technology (TU) Graz in Austria. It was conducted in collaboration with the Royal Institute of Technology (KTH) in Stockholm at the Department of Electronic, Computer, and Software Systems in Sweden. The supervisors are Professor Wolfgang Slany at TU Graz and Associate Professor Christian Schulte at KTH Stockholm. The work has been performed at the School of Information and Communication Technology (ICT) at KTH in Stockholm. Mikael Lagerkvist has acted as advisor to the work.

Acknowledgments

First of all, I want to thank my supervisors at KTH in Stockholm Christian Schulte and Mikael Lagerkvist for their support. I always received help and felt welcome and integrated at the ICT. Mikael constantly inspired me and never got tired of answering all my questions.

I also want to mention my friends from Stockholm and Austria, and my family who supported me in many different ways. Thanks to every single one of you!

This thesis would not have been possible without financial aid. That is why I want to thank the University of Technology Graz for providing me with the KUWI[1] scolarship, and *ZONTA*-Club for awarding me the *ZONTA Technical Award for Advanced Studies*.

Finally, I would like to thank Wolfgang Slany for supervising my thesis in Austria.

Thank you all!

[1] KUWI: Kurzfristige wissenschaftliche Arbeiten oder fachspezifische Kurse im Ausland

Abstract

Modeling constraint problems is a very important issue and a good model formulation is essential for finding a solution in an efficient way. However, modeling becomes difficult if the formulation process involves dealing with lowlevel issues in order to interface with the constraint solver.
The idea behind this thesis is to design a simple but expressive modeling language to formulate Constraint Programming problems independent of a solver. It should combine suitable features from other modeling languages with a new idea: The language shall contain a facility to define optional matters, like branching strategies, which are not necessarily implemented by the backend solvers.
To demonstrate the independence of the modeling language, a compiler with two backends is implemented: one translating the model to a Gecode [gcde] model, the other to a SICStus prolog [Cea95] model. The most interesting part is interfacing to two different target solvers, which might not implement all issues formulated in the modeling language, but still generating a valid model.

Zusammenfassung

Ein gut formuliertes Modell eines Constraint Programming Problems vereinfacht den Loesungsprozess ungemein. Doch Modellieren kann sehr problematisch werden, besonders wenn man dabei auch die Architektur des Solvers beruecksichtigen muss.
Diese Diplomarbeit beschaeftigt sich damit, eine simple, aber ausdrucksstarke Modellierungssprache zu entwickeln, in der man Constraint Programming Modelle unabhaengig von der Solverarchitektur formulieren kann. Diese Sprache soll wuenschenswerte Charakteristiken anderer Modellierungssprachen enthalten, aber auch eine neue Eigenschaft praesentieren: Es soll moeglich sein, optionale Kriterien zur Loesungsfindung (wie etwa spezielle Branching-Strategien) festzulegen, die aber nicht zwingend vom Solver unterstuetzt werden muessen.
Um die Unabhaengigkeit dieser Modelierungssprache zu demonstrieren, wird ein Compiler mit zwei Backends implementiert. Das eine uebersetzt in ein korrespondierendes C++ Modell fuer den Solver ”Gecode”, das andere in ein Prolog Modell fuer SICStus Prolog. Ein besonders interessanter Punkt hierbei ist, ein korrektes Modell zu generieren, obwohl nicht alle formulierten Kriterien von den Solvern unterstuetzt werden.

Contents

Chapter 1

Introduction

1.1 Aim

Designing a Modeling Language

This thesis is concerned with the design of a new modeling language, AML, to formulate constraint problems. The mathematical modeling language OPL acts as role model in the design process because of its clear and concise structure. Consequently, AML has a simple non-declarative structure, supports common expressions and relations, and offers a set of global constraints. Additionally AML introduces a completely new feature that allows to formulate optional matters. These optional matters concern the solution process (e.g. specifying the branching strategy) and need not be supported by the target solver.

Thus, the modeling language is completely independent of any solver. Obviously, this is a desirable attribute for a modeling language since it allows the user to focus on modeling rather than on interfacing with the solver's architecture or dealing with its capabilities.

Building a Corresponding Compiler

To emphasize AML's solver-independence, a compiler is built that can translate AML models either to SICStus-Prolog-models or Gecode-models in C++. These target solvers are quite different concerning architecture, offered features and some solution strategies. Gecode is a free C++ library that offers various means for extension (implementation of self-defined variables, branching or search strategies) but modeling problems implies knowledge of the system's structure since several member functions have to be implemented in addition to the problem formulation. SICStus Prolog is a solver based on the logical programming language Prolog that

provides very suitable means for modeling but has limitations concerning extendability (e.g. no self-defined search strategies are supported).

The target languages C++ and Prolog are also very different concerning their capabilities. For instance it is quite complicated to express relations that are generically sequenced by multi-nested for-loops in a declarative language like Prolog. This issue brings up a lot of interesting problems that have to be solved in a suitable way.

Furthermore, the compiler implementation is supposed to be extendable, which would allow to simply add (and remove) other target solvers.

Evaluation

Finally generated Prolog- and C++-models are tested and their performance is evaluated. Differences, limitations and possible improvements are presented and ideas for further work are introduced.

1.2 Scope

This thesis deals with the design of a modeling language so consequently there have to be limitations stated to keep the work in a sensible scope. The intention is to create an expressive but simple language that is easily extendable. Standard features such as constants, arrays, variables with ranges, for-loops and common operators are supported. Additional features, like reification or multi-dimensional arrays are not offered. These limitations restrict expressiveness and the set of problems to be formulated. However, as the language is designed to be extendable, these features are interesting to be considered for further work.

In addition to the language designed, a compiler is constructed. Again this implies some limitations. The focus of the implementation is clearly set on extendability and therefore the structure of the compiler. No effort has been set on the generation of explicitly good models or compilation-oriented issues like error-recovery.

1.3 Overview

The first part of this thesis covers background material in Chapter 2. It presents the concepts of Constraint Programming in section 2.1, gives an overview of modeling languages and OPL in section 2.2, and finally introduces the two target solvers Gecode and SICStus Prolog in section 2.3.

Chapter 3 deals with the design of the modeling language AML. It states basic requirements and describes ideas, and motivates the structure and limitations.

The compiler structure is introduced in Chapter 4. It gives an overview of the structure, functionality and motivates decisions took during the implementation. Furthermore it presents interesting challenges concerning the translation and implementation process.

In Chapter 5 typical examples are shown to illustrate and discuss interesting translation differences and resulting draw-backs. The examples are evaluated according to readability and performance of the compiler. The last part gives a conclusion to the work done and introduces some ideas on further work in Chapter 6.

Appendix A contains the user's manual for AML. It gives an introduction to the AML language on the basis of some examples. Appendix B lists the context free grammar of AML.

Chapter 2

Background

This chapter presents necessary background information for this thesis. It gives a brief introduction to the field of Constraint Programmming and its concepts (section 2.1), demonstrates issues concerning modeling languages (section 2.2) and modeling constraint problems (section 2.2.1). The modeling language OPL, that is role model for the designed modeling language, is also presented in section 2.2.2. Section 2.3 introduces both target solvers, Gecode and SICStus Prolog.

2.1 Constraint Programming

This section gives a brief overview of Constraint Programming, since a larger introduction would go beyond the scope of this thesis. The interested reader is recommended to consult [Apt03].

Combinatorial optimization problems occur in many areas and it can be difficult to find a (good) solution or even to determine, if a solution exists. Constraint Programming offers a new efficient approach to solving mathematical problems. The key issue is describing a large problem by a set of constraints. Since this approach is quite general, many areas apply Constraint Programming methods: interactive graphic systems, operation research (scheduling), molecular biology (DNA sequencing), business applications(option trading) and electrical enigneering (computing optimal circuit layouts), to mention some. The advantages of Constraint Programming are outlined by 3 basic characteristics [Apt03]:

1. **Two-Phase Approach**: The programming process consists of two clearly separate tasks: specifying the problem as a constraint model and solving it.

2. **High Flexibility**: A constraint model is very flexible, since constraints can

easily be added, removed or modified.

3. **Presence of Built-Ins**: Constraint solvers offer numerous built-in strategies and algorithms for solving constraint models.

2.1.1 Principles of Constraint Programming

Constraint Programming is a new evolving software paradigm for modeling and solving large combinatorial optimization problems. It is based upon the fact that problems can often be split into requirements, general properties, laws and restrictions - in other words - constraints. A constraint on a set of variables is defined as a relation on the variables' domains. Representing a problem by a set of constraints is called *modeling* and the resulting model is refered to as *Constraint Satisfaction Problem (CSP)*, which is defined as follows.

Definition 2.1 (Constraint Satisfaction Problem (CSP))**.** A Constraint Satisfaction Problem (CSP) consists of

- a finite set of variables $X = \{x_1, \ldots, x_n\}$ over a set of finite domains $D = \{D_{x_1}, \ldots, D_{x_n}\}$
- a finite set of constraints C, where a constraint $c_i \in C$ is a relation over a subset of variables $X_j \subseteq X$

Obviously, there exist several different possible representations of a problem as a CSP. It is important to always consider a CSP a *description* of a problem, without holding any information about how to solve it. There is a very clear separation between problem description and means to solve the problem.

2.1.2 Solving Constraint Problems

When solving a constraint problem, we want to determine the solutions to the problem in case a solution exists. Sometimes it can be interesting to question the quality of the solutions as well.

The basic process of solving starts by representing all constraints defined in the CSP by special algorithms, called **propagators**. Propagators are functions over *stores*. A store is a collection of variables that are mapped to a set of values (a subset of their corresponding domain). Example 2.1 shows a sample store.

Example 2.1 (Store)**.** *Consider the set of Variables $V = \{a, b, c\}$ with their corresponding domains $D_a = D_b = D_c = \{1, \ldots, 5\}$, then s is a store:*

$$s = \{a \to \{2, 3, 4\}, b \to \{1, 2, 3\}, c \to \{4, 5\}\}$$

Propagators are used to decrease the variables' domain space without removing any solutions. It can be considered an implementation of a constraint, as illustrated in Example 2.2

Example 2.2 (Propagator)**.** *Assume the set of variables $V = \{x, y\}$ with the corresponding domains $D_x = D_y = \{1, \ldots, 5\}$ and the propagator $p_{\geq}$ representing $x \geq y$:*

$$\begin{aligned} p_{\geq}(s) = \quad & \{\{s_x \in s(x) \mid s_x \geq min(s(y))\} \\ & \{s_y \in s(y) \mid s_y \leq max(s(x))\}\} \end{aligned}$$

The propagator is applied to the store until it reaches a fixpoint[1] (meaning it does no longer reduce the store). The following Example 2.3 demonstrates how to apply a propagator on a store.

Example 2.3 (Running a propagator)**.** *Consider the set of variables $V = \{x, y\}$ and propagator $p_{\geq}$ from Example 2.2. When applying $p_{\geq}$ on a store $s = \{x \to \{0, 1, 2, 3, 4\}, y \to \{3, 4, 5\}\}$, we get*

$$\begin{aligned} p_{\geq}(s) = \quad & \{\{s_x \in s(x) \mid s_x \geq 3\} \\ & \{s_y \in s(y) \mid s_y \leq 4\}\} = \\ p_{\geq}(s) = \quad & \{x \to \{3, 4\}, y \to \{3, 4\}\} \end{aligned}$$

If a propagator has reached a fixpoint, another available propagator is choosen to be applied on the store. This process is called *constraint propagation* and there exist many efficient strategies how to determine which propagator to pick next. Finally constraint propagation arrives at a stage, where all propagators reach a fixpoint and the store cannot be further reduced. At this point the store usually does not carry complete assignments to all decision variables (meaning no solution has been found yet).

So we can only achieve progress by dividing the problem into several disjunct subproblems and rerun propagation on them. This procedure is repeated until a solution is found or propagation results in a failure. Splitting the problem can be done by domain or by constraints. A **search tree** represents this approach: nodes contain the actual computation state that consists of a store hosting some propagators. Childnodes refer to the computation state of the subproblems and arcs to their computation. Leaves are either solutions or failures. Figure 2.1 shows a search tree for the *Send-More-Money problem* visualized by the GUI of the ALICE System [Lau78].

[1] A function $f \in X \to X$ has reached a fixpoint $x \in X$ iff $f(x) = x$.

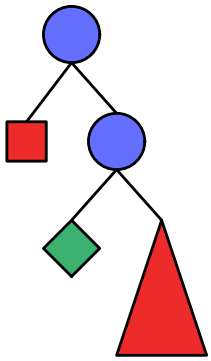

Figure 2.1: A Search Tree based on binary branching

Obviously, it is important to build the tree in a sensible way in order to gain a well-structured tree improving the search performance. There exist several **branching methods** that define a strategy on how to construct a tree. The most important issues are the choice of the variable to start the branching with (the root of the tree), and the sort of decision taken to split the problem. Mostly decisions are taken on domains, which is refered to as *domain branching*. But also the amount of pieces the problem is split into is an important issue. Deviding the problem into two (disjunct) subproblems is called *binary branching*. The searchtree in Figure 2.1 shows an example of a binary branching.

To actually find a solution in the search tree, we need to traverse it in a sensible way by applying a **search algorithm**. Generally, search algorithms are instances of simple generic iteration algorithms.

A good choice of propagators, branching and search strategy can make a big difference in the performance of solving the problem, concerning time, space and - depending on the definition - the quality of the solutions found.

Usally constraint solvers provide built-in propagators, branching methods and search strategies. The inner representation of the search tree can also affect the search time and space, but this point will be covered in section 2.3, dealing with constraint solvers.

2.1.3 Example: Sudoku

Sudoku is a very popular puzzle. It consists of a 9×9 array of fields, where each field is supposed to contain a number between 1 and 9. In every row, every column, and every major 3×3 square, each number must occur only once. An instance of

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

Figure 2.2: A Sudoku instance

a Sudoku puzzle will have some fields pre-filled with numbers. Figure 2.2 shows a sample instance of a Sudoku puzzle. According to the pre-filled fields, one has to infer the numbers to be filled in the remaining fields. This is a typical example of applying Constraint Programming methodics.

Defining the problem

We want to determine the values of all fields, so we introduce a variable for each field. The domain of each variable lies between 1 and 9, since the possible value assigned to a field lies in that range. We consider the Sudoku puzzle a matrix and set the variables' indices accordingly. We get

- a set of variables X with x_{ij} representing the field in the i-th row in the j-th column.
- a set of domains D with $d_{ij} \in D = \{1, 2, \ldots 9\}$

Now we have to define the constraints that specify the problem: In every row, column and major square, all numbers in the domain must occur exactly once. Since the cardinality of the domain equals the length of the rows and columns, and the amount of fields in a square, respectively, we can reformulate the constraint: the values assigned to the variables in every row, column and major square must be *distinct*. *Distinct* is a very popular constraint and supported by every constraint solver. We formulate the constraints as follows:

- Every field in row i must be distinct:
  c_1: $\forall i$: *distinct*$(x_{i1}, x_{i2}, \ldots, x_{i9})$ with i: $1 \leq i \leq 9$

- Every field in column j must be distinct:
  c_2: $\forall j$: *distinct*$(x_{1j}, x_{2j}, \ldots, x_{9j})$ with j: $1 \leq j \leq 9$

- Every field in square s_{kl} with $0 \leq k, l \leq 2$ and must be distinct:
  c_3: $\forall i, j$:
  distinct $(x_{ij}, x_{i(j+1)}, x_{i(j+2)},$
  $x_{(i+1)j}, x_{(i+1)(j+1)}, x_{(i+1)(j+2)},$
  $x_{(i+2)j}, x_{(i+2)(j+1)}, x_{(i+2)(j+2)})$ with $i = k * 3 + 1$ and $j = l * 3 + 1$

The first two constraints are quite intuitive, whereas the last relation is rather hard to express in a readible way. The difficulty results from the representation of the Sudoku puzzle as a matrix.

2.2 Modeling Languages

This section will present usage and advantages of modeling languages in the field of mathematical programming. It will also outline requirements of modeling languages for modeling constraint problems. Finally, the modeling language OPL [VH99] will be introduced, which was the role model for designing AML.

Modeling languages evolved in the area of *mathematical programming*, which generally deals with maximizing or minimizing an objective function, subject to constraints on variables of the function. The process of mathematical programming can be listed as a sequence of events as presented in [Apt03]:

1. Formulation of the model (containing variables, objectives and constraints)
2. Creating a problem instance by defining data (constants)
3. Generating a specific objective function on the model and data
4. Solving the problem
5. Analyzing the results
6. Refining data or model, if necessary

Modeling languages simplify solving mathematical programming problems on various aspects: generally their syntax is close to the standard way of describing mathematical problems in scientific literature, which makes it more readable and easier for modeling. For instance, the syntactic notation in OPL [VH99] for the

mathematical expression

$\sum_{i=1}^{n} a_i * x_i$ can be written as `sum {i in n} a[i] * x[i]`

Many modeling languages also provide a clear separation between the abstract model and model instances, making the model applicable to many problem instances without additional work involved. The solver is mostly considered a black-box with the modeling language acting as an interface between user and solver. This allows the user to focus on modeling the problem without bothering about low-level issues.

2.2.1 Modeling Constraint Problems

Most of the common modeling languages used for mathematical programming, provide libraries for Constraint Programming. As defined in section 2.1.1, a constraint model consists of a set of decision variables over a finite domain and a set of constraints. This section will present some basic structures and requirements to the modeling language for modeling constraint problems.

Defining Decision Variables

Decision variables can be either typed as integer, Boolean or real, even though the most commonly used are integers . To specify the decision variable's domain, we need to introduce a new data structure, which is often refered to as *range* or *domain*. Intuitively, it is defined by two values, setting the lower and upper bound of the domain. Some modeling languages require all decision variables to state a domain. For illustration, the following table shows the declaration of the decision variable x in the domain $d_x = \{1, \ldots 5\}$ in the modeling languages SICStus-Prolog [Cea95], Oz [Smo95] , OPL [VH99] , ESRA [Ag04] and the language designed in this thesis, AML.

Modeling Language	Declaration
Prolog	`X in 1..5`
Oz	`X:::1#5`
OPL and AML	`var int x in 1..5`
ESRA	`dom Dx = 1..5` `var X : Dx`

Defining Constraints

Constraints are relations over a set of variables, so most constraints are formed by relations of expressions. Some solvers make differences between linear and nonlinear expressions, for instance by offering fast algorithms for solving special linear expressions, like scalar products, for instance. Then the corresponding modeling languages provide special built-in functions for linear expressions, performing way better than the standard notation of relations over expressions would.

Most solvers provide numerous built-in constraints as well, typically complex relationships among variables that in this thesis are refered to as *global constraints*. The term *global constraint* is a very difficult one, since there does not exist a formal definition of this constraint concept. The interested reader is refered to [BVH03] for an excellent characterization of global constraints.

The most popular global constraint is *distinct*, also known as *alldifferent*.

$distinct(x_1, x_2 \ldots x_n)$ holds, when all elements x_i are assigned different values. A lot of global constraints concentrate on special areas of combinatorial optimization problems, for instance *cumulative(start, duration, resources)* is used for task ordering according to available resources in the field of scheduling.

Defining optional matters

The most important option is the specification of the branching and search strategy. Most solvers provide this option and support the most common strategies. There are also additional options that might be supported: the amount of solutions to be found or the graphical display of the searchtree that is offered by some solvers, like gecode [gcde], MOzart [MC] and alice-ml [Sys].

Requirements in a nutshell

Finally, a short summary of requirements to a good modeling language, supporting definitions of constraint problems:

Data types : support a wide range of data types, especially those that represent sets and ranges

Identifiers : ideally, a modeling language should support 4 different kinds of identifiers to be as expressive as possible: decision variables, constants, parameters and quantified variables

Constraints : the languages should offer facilities to express all kinds of relations between expressions (as far as they are supported by the solver)

Built-in Constraints : in case the solver implements some built-in constraints, like global constraints, they should be provided by the modeling language

Branch and Search Strategies : to offer the user a flexible usage of the solver, it is recommended to provide means to specialize the branching and searching algorithms implemented in the solver

There exist some additional properties that increase the expressiveness of the modeling language. One property is the ability of nesting all types to an arbitrary depth, like a set of sets of sets for instance. It is discussed in [FJMM05], which introduces the modeling language *Essence*, which is the first to provide full nesting of data types.

2.2.2 Example: The Modeling Language OPL

OPL is the role model for the language AML, since it has a lot of interesting features that will be presented in this section mostly by code-examples. The application range of OPL is very wide. It supports many areas of mathematical programming, such as linear and integer programming. This introduction will only cover features of constraint programing, for further information consult [VH99].

Basics

OPL is a very expressive modeling language, which combines features of modern programming languages and mathematical modeling languages. A model in OPL consists of an abstract part and a data part. The abstract part contains declarations of decision variables, constraints and searching strategies and therefore refers to the problem description. The data part can be stored in a separate file, containing constant declarations and corresponds to a problem instance. An abstract model can be solved according to different data files. Since the interesting issues lie in the abstract model, we will not consider the data part of an OPL model.

Basically, an abstract model consists of a declaration part, a solve part containing constraints and a search part for specifying search strategies. The declaration part covers the declaration and definition of constants, ranges and decision variables. OPL also supports the declaration of enumerate types that make the code more readable and compact. The syntactic notion is close to those of modern programming languages, as the following example demonstrates.

Example 2.4 (n queens problem)**.** *The n queens problem, based on the chess game, is to place n queens on an $n \times n$ chessboard without attacking each other. This requires that no queen may be on the same row, column or diagonal with another queen. The amount of queens is set to $n = 8$ in this example.*

```
1       int n = 8;
2       range Domain 1..n;
3       var int queens[Domain] in Domain;
4
5       solve {
6          forall(ordered i, j in Domain) {
7            queens[i] <> queens[j];
8            queens[i] + i <> queens[j] + j;
9            queens[i] - i <> queens[j] - j;
10      };
11
12      search {
13          forall(i in Domain)
14            generate(queens[i]);
15      };
```

The example presents the clear structure of OPL models and some powerful features. The most interesting feature is the *forall* construct in line 6 that generates the sequence of constraints

```
queens[i] <> queens[j];
queens[i] + i <> queens[j] + j;
queens[i] - i <> queens[j] - j;
```

for all $1 \leq i < j \leq 8$. This construct allows to express a sequence of constraints very compact and readable. Another interesting point is the specification of the search strategy in line 12, where `generate` uses the *first-fail principle* to search for a solution on every element of the array `queens`. This search strategy can be described as *"start, where you are most likely to fail"*, meaning that the variable with the fewest possible remaining alternatives is chosen to start the search on. Aside of providing common search strategies, OPL also allows the user to define his own search procedures that can be completly adapted to the problem.

High-Order Constraints

High-Order constraints are better known as *reified constraints* that are logic constraints on a set of constraints. This is done by assigning Boolean variables to a set of constraints and setting them in relation to each other. Obviously, this method allows us to express constraints very efficiently, which is illustrated in the next example.

Example 2.5 (magic sequence)**.** *A magic sequence is a sequence $S = (s_0, s_1, \ldots s_{n-1})$ of n numbers, where the element s_i represents the number of occurences of i in the sequence S. For instance, the sequence $S = (2, 1, 2, 0, 0)$ for $n = 5$ is a magic sequence, since 0 and 2 occur twice, 1 occurs once and 3 and 4 occur zero times.*

```
1       int n = ...;
2       range Index = 0..n-1;
3       range Domain = 0..n;
4       var Domain s[Index];
5
6       solve {
7          forall(i in Index)
8            s[i] = sum(j in Index) (s[j] = i);
9       };
```

The expression `sum(j in Range) (s[j] = i]))` in line 8 is used to count the amount of variables in `s` for which `s[j] = i` holds. For solving this expression, the solver replaces each constraint in the expression by a Boolean variable that is assigned the value 1 if the constraint is satisfied, and 0 if not. Obviously, high-order constraints are powerful tools for expressing constraints in a concise way, keeping the constraints comprehensible. This example also demonstrates the usage of an external data file: the length of the magic sequence `n` is read from a data file, which is expressed by the notion `int n = ...;` in line 1.

2.3 Constraint Solvers

This section will give an introduction and overview of the constraint solvers gecode [gcde] and SICStus Prolog [Cea95], since these are the corresponding solvers for the models AML can be translated to. These solvers and their modeling languages are highly different and therefore very interesting to compare and examine.

One important issue is the inner representation of the search tree, since both solvers use completely different strategies.

Search Tree Representation

As discussed in section 2.1.2, in order to solve a problem, the solver has to perform search, where the searching mechanism is represented by a search tree. When traversing the tree for a solution, it is common to end up in a branch, where the search fails. Then it is necessary to get back to a good point of the search tree and find another path, which may lead to a solution. This can be done by applying one of the following methods:

1. Memorizing states by **copying** them
2. Reconstructing states by either **trailing** or **recomputation**

Recomputation regenerates the states from scratch, while trailing reconstructs the state by always recording the changes that have been applied on the state and simply redoing them. Copying uses a very different approach: when computing a new state, first the old state is copied and then propagated on. So every traversed state is available at any time.

Consequently, systems based on trailing deal with *operations* on data structures, while copying-based systems are only concerned with *data structures*, which makes search independent of operations on data structures for copying.

Most solvers are trailing-based and only few use copying. However, [Sch99] gives an excellent comparison of trailing-based and copying-based systems, demonstrating their competitiveness.

2.3.1 Gecode

Gecode stands for *generic constraint development environment* and is a free system for developing constraint-based systems and applications. It is a very young project, which has first been released in December 2005. The developers consist of a small group of researchers at KTH Stockholm in Sweden and Saarland University in Germany. The development of gecode was motivated by the idea of building a flexible, open and free constraint solving system that is applicable for academic pursuit of Constraint Programming. Some of the developers could also contribute their experience gained by developing the MOzart System [MC].

Features of gecode

Gecode is a C++ library with various interesting features. It is an open system that can easily be interfaced to other programming environments and supports the implementation of new propagators, branching strategies, search engines and also new variable domains. Gecode is distributed on a BSD-style licence, and offers free source code, documentation and examples for download. It is portable since it holds the C++ standard and can be compiled with common C++ compilers and runs on several machines. Benchmarks presented at [gcde] show that Gecode is efficient according to time and memory usage and therefore competitive with other common constraint solvers.

Being a C++ library, the main focus is set on solving rather than modeling: models for gecode have to be written in C++ and adapted to the solver structure, which includes, for instance, defining several member functions for classes used.

So modeling a problem also involves understanding the structure of the solver, which is time consuming for the unexperienced user. But currently a Java interface is under development that is supposed to simplify modeling.

The structure is very beneficial when it comes to implementing self-defined variable types, branching or search strategies. New variable domains can even be implemented having the same level of efficiency as integer set or finite domain variables.

Search is based on the rather unusal strategy of copying together with recomputation. Therefore every problem model has to provide a copy method in order to be solved.

Gecode offers common constraints, such as arithmetic, boolean, linear equations and various global constraints. Since gecode provides finite integer set variables, it also offers set relations, set operations, and special set constraints.

2.3.2 SICStus Prolog

Prolog is a logic programming language developed in the early 80s and is now part of the International Standard[2]. There exist various versions of Prolog and one of them is SICStus Prolog, developed by the Swedish Institute of Computer Science (SICS). SICS is a non-profit research organization that first focused on parallel logic programming and switched research to the area of constraint programming.

Features of Prolog

SICStus Prolog comes with 3 constraint solvers who are based on the general constraint logic programming scheme: a solver for Boolean constraints, for constraints on reals [Hol95] and for finite domain constraints [COC97]. We will only consider the solver for finite domain constraints.

Prolog is a very expressive modeling language, especially for modeling constraint problems, since it has a logic and simple structure. As the constraint solver of SICStus Prolog can be considered a part of the modeling language, the focus is clearly set on modeling, rather than on extending solving mechanisms.

The system has two classes of constraints: primitive and global constraints. Numerous global constraints are provided and reification (high-order constraints) is also supported for primitive constraints.

Unfortunately, SICStus Prolog is only partially extendable: It is possible to implement new domain variables and constraints (both primitive and global), but there are no means to construct self-defined search or branching strategies. SICStus Prolog uses trailing for search.

[2] ISO/IEC 13211-1

2.3.3 Gecode vs. SICStus Prolog

The biggest difference between gecode and SICStus Prolog lies in gecode being a C++ library and SICStus Prolog a logic programming system where the solver is included in the language's structure. This naturally characterizes SICStus Prolog as being better for modeling problems and gecode being more extendable.

Both solvers use completely different strategies for searching solutions: gecode is copying-based and Prolog applies trailing.

There exist some benchmarks comparing gecode to SICStus prolog according to runtime on some model problems.

Chapter 3

Designing AML

This chapter presents the design process of AML (A(ngee's) Modeling Language) and focuses rather on the ideas and motivations during design than the language itself. For a detailed description of the language, see the user's manual in Appendix A. AML's grammar is listed in Appendix B.

3.1 Motivation

Before designing AML I tested some typical mathematical modeling languages, because I wanted to chose a role model for AML. The most inspiring language turned out to be OPL (discussed in section 2.2.2): it has a very concise model structure and its syntax is very close to those of modern programming languages. So AML combines some useful features of OPL with some new ideas on a rather simple level.

3.2 Goals and Requirements

I designed the structure by first collecting a list of **basic requirements** for the language to model constraint problems.

Basic types : Most constraint problems are integer-based and to keep things simple, I decided to only provide integer typed variables. Still I wanted to stress that the language is strongly typed, so when declaring a variable the keyword `int` is necessary.

Arrays : I decided to provide arrays instead of lists, tuples or records, since I wanted the language to be explicitly non-declarative. Only only one-dimensional

arrays are supported, since further nesting would have gone beyond the scope of this thesis.

Constants : I wanted to support constants to make it possible to describe the model in an abstract way. It was also important for me to separate the constant declaration from the rest of the model in order to obtain a clear structure. This is why constants are to be defined in a destined data block that is mainly inspired by the model structure of the OPL language.

Decision Variables : Decision variables are either integer arrays or simple integer variables.

Ranges : Ranges are set during the declaration of variables, but there does not exist a specific range type, since I wanted to have as few types as possible and ranges are not necessarily needed as types to model constraint problems.

Relations : Relations are binary over arbitrary linear or nonlinear expressions, since this is the simplest way to describe relations. First I was uncertain, if I should differentiate between linear and nonlinear expressions, but that would have made it more complicated for the user to model constraints.

Global Constraints : I wanted to provide some basic global constraints that are commonly provided by solvers and decided to support *distinct*, *serialized* and *cumulative*.

Constraint Sequencing Sequencing constraints generically is a very powerful method to define a set of constraints, since the formulation is simple and readable. This is why I wanted to offer a `forall` construct, which can be arbitrarly nested. The syntax of the forall loop was also inspired by OPL.

After stating basic requirements, I formulated some goals of the language: a very important goal is to obtain a clear and concise model structure. That is why I decided to construct the model in form of a block structure, to separate constant declaration, decision variable declaration and constraints. I also introduce an option block, which allows the user to state some special matters, like branching strategies. The idea of an optional block is new and interesting: it is supposed to define some issues, which do not have to be supported by the solver that the model is interfaced to.

I also wanted the syntax to be close to modern programming languages, since standard users of modeling languages are usually familiar with programming.

3.3 The Language Structure

As mentioned, an AML model is constructed of blocks. This section will give details on the motivation and decisions taken in order to form the structure.

Every model is initiated with the key word `model` followed by a modelname, identifying the model. Then braces { } form the model block that contains the model description. I have chosen braces to enclose blocks, because I believe that it is the most readable way.

A big advantage of the usage of blocks is that an order of constant, variable and constraint declaration is implied in the structure. This makes it easier to compile and process the model for the programmer on the one hand, but also easier for the user to write correct code on the other hand.

A model in AML is structured by the following hierachy:

1. data declaration block (optional)
2. variable declaration
3. constraint declaration block
4. options block (optional)

The order of the blocks is inspired by OPL. In the following subsections, each block will be discussed.

Data Declaration

As mentioned above, I wanted to support constant declarations and separate them from the rest of the model in a block. This approach is inspired by considering constant data as problem instance. A possible extension would be allowing multiple data blocks and pick one of them for processing. This would also come close to the OPL approach that can load constant declarations from different files.

Constraint Declaration Block

The constraint declaration block contains all constraints and is initiated by the keyword `solve`, as in OPL. Collecting all contraints in a block makes the problem formulation more readable.

Options Block

The option block is the most interesting part of the language. It contains statements specifying special issues that concern the solving process. The grammar is very vague concerning option statements: every statement of the form

```
IDENTIFIER = EXPRESSION ;
```

is allowed, but only known options are processed by the compiler. Every unknown statement is simply ignored and results in a warning message of the compiler. This brings a lot of advantages for the modeling process: the user does not have to bother whether the options he choses are supported by the target solver. Additionally the user is always informed about which options have been processed and which are not applicable. This makes the options block extremly userfriendly.

These advantages are the main reasons, why I have chosen to specify the options block in this way. Another important reason is the requirement of the language being independent of the target solvers, which is hereby clearly demonstrated.

3.4 Limitations

This section discusses some modeling boundaries and why they exist. It motivates some decisions I took in designing the language and discuss some possible extensions.

When designing AML, I knew that it would be a simple language and had to decide which limitations to accept and which not. I wanted to get as much expressiveness as possible with the smallest effort, since I did not want to let constructing the AML compiler get out of bounds.

Only One-dimensional arrays : One of the first decisions I took, was to restrict arrays to be one-dimensional. Supporting multi-dimensional arrays would have been more expressive, since it would allow to represent matrices and related structures very efficiently. Still I found the effort in implementation would have been far higher than the expressiveness gained by introducing multi-dimensional arrays.

No datatype for domains : No specific type representing domains or ranges exist, even though it is common in other modeling languages for constraint programming. I decided that there was no necessity for adding a range type, as it affects the language's expressiveness only barely, since AML is very

simple. It would have been more necessary, if the modeling language was larger.

No reified constraints : It is not possible to formulate reified constraints (high order constraints) in AML. I actually did not think of implementing high order constraints when designing AML. It would certainly be an interesting feature to add to AML.

Only few global constraints : AML offers only three global constraints. Intutively, this is because supporting all possible global constraints would have gone beyond the scope of this thesis. So I picked three popular global constraints that both target solvers support.

Chapter 4

Compiler Structure

This chapter presents the structure of the compiler implementation on an abstract level. It presents ideas, motivation and problems during the implementation process. The detailed functionality of the compiler is discussed in context with the structure. Basic knowledge of compiler construction is presumed, since an introduction would go beyond the scope of this thesis. The so-called *dragonbook* [ASU86] gives an excellent introduction and overview of compiler construction.

4.1 Overview

The compiler implemented in this thesis is supposed to translate an AML model into a corresponding Gecode or SICStus Prolog model. Figure 4.1 shows the basic structure of the compiler implementation: given an AML file, the frontend translates it into a model description, defined in the intermediate language XML. This model specification is then applied to the backend that translates it either to a Gecode or SICStus Prolog model.

Frontend and backend are implemented independently and work stand-alone. This is an important issue concerning re-usability and extensionability, since the structure easily allows to remove or add components (like another backend) to the compiler.

The following sections present the compiler parts in more detail.

4.2 Frontend

The frontend has been implemented in *ocaml* [Rém00]. Being a functional programming language, ocaml is very expressive, provides pattern matching and sup-

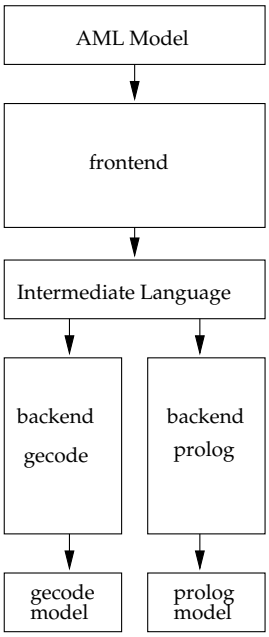

Figure 4.1: The basic compiler structure

ports even imperative and object oriented styles. There also exist libraries for lexer and parser generation, which are based on the popular tools *lex* and *yacc* [LMB92].

ocamllex allows to specify tokens by regular expressions and generates a corresponding lexer. *ocamlyacc* constructs a LALR parser, according to a context free grammar provided. Both tools can be interfaced and easily integrated in ocaml code.

4.2.1 Functionality

The frontend takes an AML model as input and produces a XML description of the model. The transformation process can be described by the following sequence of tasks:

1. **Read input file:** read the input file from the arguments passed
2. **Generate tokens:** the lexer filters the input file and splits its content into a sequence of tokens.
3. **Parse the grammar:** the parser takes the list of tokens as input and parses it according to the specified grammar.

4. **Construct an Abstract Syntax Tree (AST):** in case the parser processes the tokens sucessfully, it generates an AST. ASTs represent the grammar by a tree-structure: leaves correspond to terminals and nodes to non-terminals.

5. **Process the AST:** semantic features are checked (such as valid identifier declarations), declared constants are inserted into expressions, which are evaluated afterwards.

6. **Transform AST to Intermediate Specification:** finally a new tree-like data structure is generated according to the information stored in the AST. This data structure corresponds to the intermediate language.

7. **Write Intermediate Specification into a File:** By applying a simple to_string method the intermediate specification is stored in form of a string that is written in a file.

Constant Processing

An important issue is the processing of the constant declarations: all constants defined in the data block are inserted in the expressions they appear in. Therefore the model description in the intermediate language corresponds to a model instance and carries no data block.

Error Handling

Generally, if an error is detected, a message is printed on the standard output, containing information about the type of error and additional information, if available.

Errors in the lexer or parser are treated immediately, while the other modules handle errors by exceptions. I did not apply any error recovery and did not focus on error handling that much, since it would have gone beyond the scope of the thesis.

4.2.2 Implementation Structure

The implementation consists of a lexer, a parser and 3 additional modules.

Lexer and Parser

The lexer is generated by `ocamllex`, which takes a description of tokens by a special list of regular expressions. This list may also contain user-defined functions or statements. `ocamlyacc` creates the parser according to a specification file containing a list of terminals, grammar rules, operator precedences and user-defined functions. It also specifies the structure of the AST (abstract syntax tree) in the

grammar rule definition: every rule states the node or leaf type which is returned, in case the rule holds.

Additional Modules

The first module `ast` contains the data structure definition and functions for the AST representation. The second module `constant` contains functions for constant processing, such as insertion in expressions and expression evaluation. Finally, the module `xml` contains the data structure definition and functions for the intermediate language that is XML.

4.2.3 Challenges in the Design Process

The first part in designing the frontend was the definition of the AML grammar in the parser, which is listed in Appendix B. The grammar should be simple, easily extendable and strict. Since I did not have any experience with the design of languages, it was challenging and difficult. Consequently, I discovered opportunities for improvement after the whole implementation process. This is why I want to discuss challenges during the design of the AML grammar in this section.

My appraoch was to structure the grammar by simple means and keeping it as restrictive as possible. That seemed to work, since I only made few changes during the whole implementation process. Still I did not consider an important point: making some parts more abstract can increase extendability without affecting the strength of the grammar. The following parts could have been implemented in a better way:

- Global constraints:
  Global constraints are treated as keywords in the grammar. So extending global constraints inforces extending the grammar. It would have been a better approach to treat global constraints in a more abstract way, like function calls. Then extensions would not imply extending the grammar, but simply extending transformation rules in the frontend and backend part.

- Options:
  Options are defined as expressions that are assigned to identifiers. This makes it difficult to add options taking more than one parameter, like defining branching strategies for a specific decision variable. Such an extension would also affect the grammar. Hence it would again have been better to define the options-grammar more abstractly in form of function calls that can take multiple parameters.

```
<person>
   <name>
       <firstname>Andrea</firstname>
       <lastname>Rendl</lastname>
   </name>
   <age>23</age>
</person>
```

Figure 4.2: A sample XML description of a person

I unfortunately overlooked the opportunity to represent some parts in the grammar by function calls, since modeling languages are commonly not concerned with functions.

4.3 Intermediate Language XML

The intermediate language should be simple and store relevant information in a concise and well-structured way. In addition, it should also be readable and there should exist means to verify and process its representation. I have chosen XML to be the intermediate language, since it conforms to all requirements.

XML stands for *Extendable Markup Language* and provides means to represent data in a tree-like structure. An XML description consists of a set of elements, whose content can either be another element or some data. This implies a hierarchy in the information described, which makes XML documents very readable (assuming a reasonable structure).

As demonstrated in the example shown in Figure 4.3, XML is very simple and easily comprehensible. That's why XML is a very popular markup language and most programming languages provide libraries for parsing XML documents. There also exist a lot of facilities to verify XML docmuents and display elements in a fancy way.

The structure of the XML document representing the AML model is strictly stated by a DTD document `model.dtd`, which is used to verify the document in the backend structure. The file is included in the compiler.

4.4 Backend

The backend part consists of two backends; one translating to C++, the other to Prolog. Both backends have much in common and the structure has been implemented on a very abstract level in order to simplify extensions, like adding another target solver.

The implementation language for the backend is Java [1.5]. First I wanted to implement the backend in ocaml, but had problems with the xml-light library, providing an ocaml XML parser. I finally decided to switch to Java that also provides XML parsers.

4.4.1 Functionality

The backend takes an input file and a flag stating the language to translate to. The input file is assumed to be a XML document, following the rules of the DTD specification stated in `model.dtd`, which comes with the compiler. The correctness of the model structure is also assumed, since backend-independent semantic checks (for instance check for non-ambigious variable names) are done in the frontend.

Both backends have the same structure, but intuitively their functionality differs in some parts. The Gecode Backend is concerned with generating a C++ class where the constructor holds the problem description. Several member functions have to be generated to conform to the solver's class hierarchy. The backend translating to Prolog generates a predicate that contains the problem description and brachning strategy.

Section 4.4.3 presents some interesting challenges in the translation process for both backend types.

4.4.2 Implementation Structure

XML parsers

The most important issue of the backend was to build it in a structure that is easily extendable.

There exist two kinds of XML parsers: SAX (serial access) parsers that traverse the XML document and give information about the currently processed XML-element. The alternatives are DOM (document object model) parsers that generate a tree-like representation of the XML document. I picked the SAX parser, since its approach is simpler, faster, less memory-intensive and sufficient for the needs of the backend.

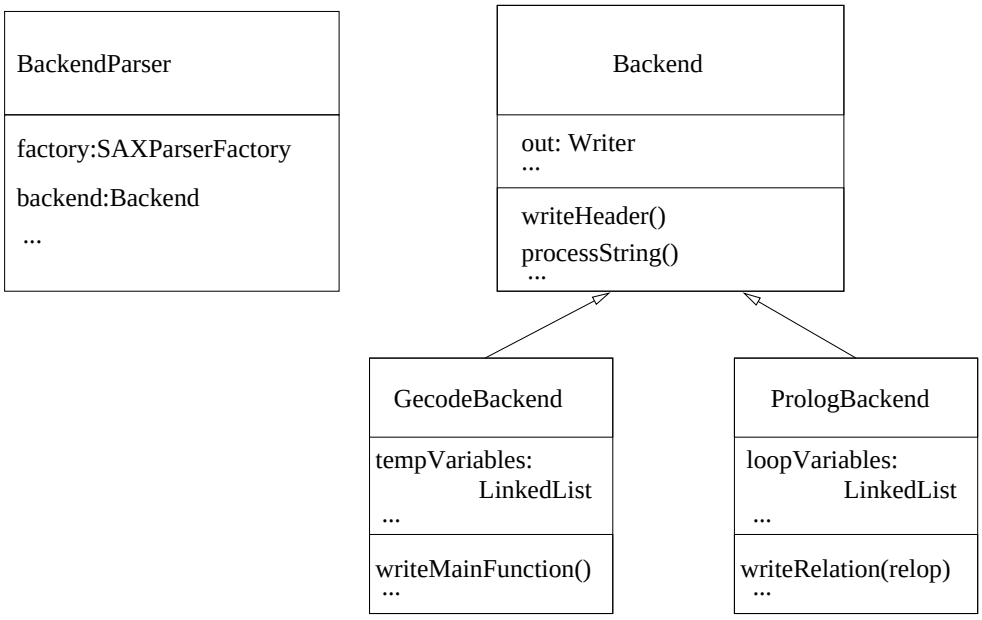

Figure 4.3: The simplified class structure of the backend

Class Structure

SAX parsers do not store information about the XML file they parse, but notify about the currently parsed element. So immediate action has to be taken during the parsing process. That is why I decided to build the whole structure around the SAX parser, and process the XML element content according to the target-backend specified. Figure 4.3 shows the skeleton of the major class structure.

BackendParser : The basic part of the backend is the class BackendParser, which is a subclass of DefaultHandler. Its constructor takes a XML file and a backend-flag, denoting whether to translate to Prolog or to C++. The field `backend` of the type Backend, performs the main translating process and states how to handle currently processed XML elements.

Backend : The abstract class Backend declares fields and methods necessary to interface with the SAX parser in BackendParser. It also contains the field `out` of type Writer which is the output file. Any feasible backend-specification has to inherit from this class.

GecodeBackend : GecodeBackend is a subclass of Backend and the implementa-

tion of the backend translating from XML to C++.

PrologBackend : GecodeBackend is a subclass of Backend and the implementation of the backend translating from XML to Prolog.

Functionality

The backend works as follows:

1. The main class creates a BackendParser instance, depending on the target language to translate to (which is read from the input arguments).
2. The BackendParser instance creates an Backend instance corresponding to the target language.
3. BackendParser starts the parsing process on the XML file. On every XML-element it calls Backend methods that set flags or process the XML element's content.
4. After the XML file is processed, `out` of type Writer (that stores the translated model) from Backend is flushed into the file.

This structure makes it very easy to extend the backend and add other target solvers, by simply creating another subclass of Backend.

4.4.3 Challenges in the Design Process

Handling Non-linear expression in Gecode

One feature of Gecode is, that there is no mean to state arbitrary relations between nonlinear expressions. This means, that a relation can only be over linear expressions. Hence nonlinear expressions have to be stated separately and represented by a temporary variable. This temporary variable is then inserted in the relation, since a single variable is always linear.

Consider the relation `a*b > c` which contains the nonlinear expression `a*b`. This expression has to be stated separately and assigned to a temporary variable.

```
tmp = a * b
```

This variable can now be inserted for the nonlinear expression, making it linear on the relational level.

```
tmp > c
```

Creating temporary variables introduces other issues: temporary variables have to be defined before they are used in the C++ model. Since one never knows, if the upcoming expressions are nonlinear, some information has to be buffered.

Translating For-loops to Prolog

Prolog is a non-declarative logical programming language and does not support for-loops. AML offers for-loops and consequently, a strategy has to be found to translate relations defined in (multi-nested) for-loops from AML to Prolog. Basically, a for-loop allows to specify a sequence of relations in a generic way. A possible strategy is to generate the corresponding sequence in Prolog, which leads to unfolding of the for-loop. The following code fragments illustrate how a for-loop is unfolded.

AML Code	Generated Prolog Code

`for(i in 0..3) {`
`  x[i] == i;`
`}`
	`X0 #= 0,`
	`X1 #= 1,`
	`X2 #= 2,`
	`X3 #= 3`

The example above shows a simple version of a for-loop. However, when it comes to multi-nested for-loops with dynamic indices containing expressions the automated unfolding process becomes more challenging as the following example demonstrates.

AML Code	Generated Prolog Code

`for(i in 0..2) {`
`  for(j in i+1..2) {`
`  x[i] != x[j];`
`  }`
`}`
	`X0 #\= X1,`
	`X0 #\= X2,`
	`X1 #\= X2`

Chapter 5

Examples and Evaluation

5.1 Motivation

It is interesting to compare generated models, because some solver-dependent aspects are treated completely differently in the translation process. Comsequently, the quality of the generated models can differ quite drastically.

This sections presents some typical examples, which are translated and solved by both solvers. The generated models are evaluated according to some specified criteria. This demonstrates differences and weaknesses in the translation process and how they affect the generated model and its performance. The evaluation underlies a set of criteria, which are

- performance (space, time)
- readability
- limitations for larger problems

5.2 Example: nQueens

5.2.1 Overview

The nQueens problem is placing n queens on a n $\times$ n chessboard without attacking each other. So no queen may be on the same row, column and diagonal as another. The model description in AML is presented below.

```
1 model QueensN {
2   data {   /* amount of queens  */
3     int n = 8;
```

```
4   }
5
6   var int queens[n] in 0..n-1;
7
8   solve {
9    /*  no queen may be on the same column as another */
10    distinct(queens);
11
12    forall(i in 0..n-1) {
13      forall(j in i+1..n-1) {
14         queens[i]+i != queens[j] + j;
15         queens[i]-i != queens[j] - j;
16      }
17    }
18  }
19 }
```

nQueens is a very interesting example to examine. On the one hand, the problem size can be arbitrary altered by the parameter `n`. On the other hand, the problem description contains forloops that are treated differently in the translation process since Prolog does not offer a `forall` construct. Consequently, every forloop stated in the AML model is unfolded in the Prolog model. The following code segment shows the unfolded forloop of the generated Prolog file for the smallest possible value `n = 5`.

```
...
(Queens0__a + 0) #\= (Queens1__a + 1),
(Queens0__a + 0) #\= (Queens2__a + 2),
(Queens0__a + 0) #\= (Queens3__a + 3),
(Queens0__a + 0) #\= (Queens4__a + 4),
(Queens1__a + 1) #\= (Queens2__a + 2),
(Queens1__a + 1) #\= (Queens3__a + 3),
(Queens1__a + 1) #\= (Queens4__a + 4),
(Queens2__a + 2) #\= (Queens3__a + 3),
(Queens2__a + 2) #\= (Queens4__a + 4),
(Queens3__a + 3) #\= (Queens4__a + 4),
(Queens0__a - 0) #\= (Queens1__a - 1),
(Queens0__a - 0) #\= (Queens2__a - 2),
(Queens0__a - 0) #\= (Queens3__a - 3),
(Queens0__a - 0) #\= (Queens4__a - 4),
(Queens1__a - 1) #\= (Queens2__a - 2),
```

```
(Queens1__a - 1) #\= (Queens3__a - 3),
(Queens1__a - 1) #\= (Queens4__a - 4),
(Queens2__a - 2) #\= (Queens3__a - 3),
(Queens2__a - 2) #\= (Queens4__a - 4),
(Queens3__a - 3) #\= (Queens4__a - 4),
 ...
```

5.2.2 Evaluation

Obviously, unfolding the forloop easily becomes a problem when parameter `n` increases. This not only affects readability and model size, but also compilation time drastically. The following table illustrates the differences in compilation time with an increasing `n`. Time measurement has been done by the help of the program `time`.

n	solver	total time [sec]	user CPU time [sec]	system CPU time [sec]
15	Gecode	0.519	0.438	0.047
	Prolog	0.638	0.482	0.055
20	Gecode	521	0.436	0.045
	Prolog	0.624	0.546	0.069
25	Gecode	0.521	0.0408	0.076
	Prolog	0.729	0.639	0.082
30	Gecode	0.520	0.418	0.061
	Prolog	1.446	0.892	0.096
35	Gecode	0.520	0.417	0.068
	Prolog	1.750	1.525	0.137
40	Gecode	0.520	0.403	0.074
	Prolog	3.076	2.842	0.172
45	Gecode	0.521	0.412	0.067
	Prolog	5.366	5.063	0.248
50	Gecode	0.571	0.438	0.048
	Prolog	9.860	9.416	0.391
55	Gecode	0.521	0.404	0.077
	Prolog	16.340	15.601	0.525
60	Gecode	0.520	0.409	0.070
	Prolog	32.556	30.988	0.806

5.3 Example: Grocery

5.3.1 Overview

Two children buy 4 items at the grocery. The cashier erroneously multiplies the prizes instead of adding them, but still gains the same result, which is 711. The AML model formulating *grocery* is presented below.

```
1 model Grocery {
2
3  data {
4    int prize = 711;
5  }
6
7  var int items[4] in 0..711;
8
9  solve {
10   item[0] + item[1] + item[2] + item[3] == prize;
11   item[0] * item[1] * item[2] * item[3] == prize*100*100*100;
12
13   item[0] > item[1];
14   item[1] > item[2];
15   item[2] > item[3];
16  }
17 }
```

This example is interesting, because it contains a relation between nonlinear expressions. Unfortunately it is not scalable.

5.3.2 Evaluation

Non-linear expressions in relations are treated differently when translating to Gecode than when translating to Prolog. As Prolog supports relations over non-linear expressions, the information structure does not have to be manipulated. But Gecode only offers facilities to formulate relations over linear expressions. Thus it is necessary to introduce temporary variables to linearize the structure of expressions. This increases the amount of variables. Consequently, a high amount complex non-linear expressions can affect the quality of the generated model.

As the example is not scalable, this effect cannot be demonstrated like in the nQueens example. The following table shows the compilation time for both models.

Solver	total time [sec]	user CPU time [sec]	system CPU time [sec]
Gecode	0.471	0.379	0.066
Prolog	0.471	0.387	0.050

Chapter 6

Conclusions and Further Work

This thesis has presented a new modeling language AML with its corresponding compiler. This chapter will give some conclusions about the work and present ideas for further work.

6.1 Conclusions

The first part of work on this thesis dealed with the design of the modeling language AML. After investigating several standard modeling languages, OPL has been chosen to act as role model for AML, which turned out to be a good choice. AML's syntax is simple and strictly structured. The set of supported global constraints and options was specified and the context-free grammar for AML was stated.

The second part involved implementing a corresponding compiler for AML where the focus was set on extendability. This allows to easily add other target solvers to the compiler.

6.2 Further work

A very interesting idea for further work is extending AML. It is a simple language and can express a certain set of problems. However, the set of expressable problems could easily be augmented if AML would support multi-dimensional arrays, which would allow to represent matrices and other more complex structures. In addition, means for reification could be added and the set of supported global constraints could be expanded.

The AML compiler translates AML models to C++ and Prolog and is easily extendable. Thus, adding other target solvers like Oz [Smo95] can be interesting. However, the most challenging point for further work is setting the focus on generating *good* or even *optimal* models, which has not been considered in this thesis.

Appendix A

User's Manual

This manual documents the AML modeling language, which is used to model constraint problems and the AML compiler, which can translate the model into either a C++ model adapted to Gecode, or a Prolog model for the solver integrated in SICStus prolog.

A.1 Introduction

AML is a modeling language for modeling constraint problems. It is inspired by the OPL language and has a close syntax. AML is strongly typed and only supports integer variables and one-dimensional arrays. AML allows you to focus on the modeling process without bothering about how to interface with the solver. Still it is possible to specify options, like search and branching strategies for a solver, which are ignored if the solver does not support them.

A.2 The AML model structure

An AML model has a very strict and concise structure: Every model is defined in a `model` block with an user-defined model name:

```
model MODELNAME {
   /*  model definition  */
}
```

A model consists of blocks in order to separate constant delarations, variable declarations, constraint declarations and options. Comments can be written in `/* a comment */`.

A.2.1 The Data Block

The first block is the `data` block, which is optional and used to define constants. Constants are integer variables or integer arrays that have to be fully defined. The following code shows some sample constant declarations.

```
model Test {

  data { /* constant declarations */
     int b[2];
     int a = 4;
     b[0] = a + 2;
     b[1] = b[0]*5;
  }
  /* variable and constraint declaration */
}
```

If a constant is assigned multiple values, the last value assigned is finally mapped to the constant.

A.2.2 Decision Variable Declaration

Decision variables may be integer variables or arrays and are denoted by the keyword `var`. Domains may be defined, but only in the context of a decision variable declaration; there exists no independent domain type.

```
 model Test {

  data {
    int n = 9;
  }

  /* variable declaration */
  var int x in 1..n;
  var int y[4];
  var int z[9] in 1..4;

  /* constraint declaration */
}
```

The code sample shows some possible variable declarations. The domain is defined by a lower and an upper bound and denoted by the expression `in LOWERBOUND`

`..` `UPPERBOUND`. There may not exist variables and constants with the same name. Intuitively, a valid model contains at least one decision variable.

A.2.3 The Constraint Block

Constraints are defined in a constraint block, initiated by the keyword `solve`. There are two types of constraints: a binary relation over linear or nonlinear expressions (the syntax of the relational operators can be found in the AML grammar description in Appendix B) or a global constraint. AML supports the following global constraints:

1. `distinct(x)` where `x` is a declared decision variable array
2. `serialized(start, duration)` where `start` and `duration` are declared decision variable arrays
3. `cumulative(start, duration, resources, limit)` where `start`, `duration`, `resources` are declared decision variable arrays and `limit` is an integer or integer variable.

The following sample code shows how to use global constraints in a simple scheduling problem.

```
model Scheduling {

  var int s[2] in 0..2;
  var int d[2] in 1..3;
  var int r[2] in 1..4;

  solve {
     cumulative(s,d,r,2);
     s[0] < 2;
     s[1] > 0;
     d[0] >= 2;
     d[1] > 0;
     d[1] < 2;
     r[0] >= 2;
     r[1] > 0;
     r[1] != 2;
     r[1] != 3;
 }
}
```

Additionally, it is possible to generate a sequence of constraints by using the `forall(i in min..max)` construct, where `i` is an arbitrary identifier and `min` and `max` are integers. If the value of `min` is greater than `max`, the relations defined in the forloop will simply be ignored. Forloops can be arbitrarily nested. Loop-variables are only known in the scope of the forloop. The n-queens problem demonstrates the usage of forloops.

```
model Queens {
    data {     /* amount of queens  */
       int n = 8;
    }

    var int queens[n] in 0..n-1;

    solve {
      distinct(queens);

      forall(i in 0..n-1) {
         forall(j in i+1..n-1) {
            queens[i]+i != queens[j] + j;
            queens[i]-i != queens[j] - j;
         }
      }
   }
}
```

A.2.4 The Option block

The option block is optional and allows you to specify special features, like search or branching strategies. In case an option is not supported by the compiler, it is simply ignored. AML supports the following option definitions:

Option	Description	Supported by
branching_var = OPTION;	OPTION defines which variable to choose for branching (first fail as default)	Gecode Prolog
branching_val = OPTION;	OPTION defines which value to choose for branching (minimum as default)	Gecode
solutions = NUMBER;	NUMBER specifies amount of solutions to be searched for	Gecode
iterations = NUMBER;	NUMBER specifies amount of iterations to be searched for	Gecode

The option `branching_var` is the most complicated one, since Gecode and SICStus Prolog have different branching options to chose from and only some overlap. Still, the compiler is very helpful: stating an unknown or ambigious option results in a warning message from the compiler providing information about possible alternatives.

A.3 Installation of the compiler

This section covers the installation process of the AML-compiler. Please note, that the system was built for usage in Unix-systems. In order to compile the frontend and backend, you will need a version of `ocaml 3.08` (or newer) containing the `ocamlyacc` and `ocamllex` tools and at least `Java 2 Platform Standard Edition 5.0`.

First move the file `AMLCompiler.tar.gz` to an existing directory. Change to that directory and decompress the file by typing

```
tar -zxvf AMLCompiler.tar.gz
```

in your shell. This will create the directory `AMLCompiler`. It contains two bash-script files to build the compiler: `build_all` and `build_clean`. Simply run one of them by typing

```
./build_all
```

in your bash-shell and the AMLCompiler is built and installed. Further details can be found in the `README` file.

A.4 Running the compiler

The compiler has several features and translates in several ways. Make sure, that a copy of the file `model.dtd` (which describes the XML structure of the constraint model) is in the same directory as your AML-model. The file can be found in the `examples/` directory.

1. Translating AML to C++ GECODE
   To translate an AML model to a C++ Gecode-adapted model, run the script `aml2cc` in a bash shell:

   ```
   $ ./aml2cc inputfile
   ```

   where `inputfile` is a AML-Model. The generated C++ file will be written into `inputfile.cc`.

2. Translating AML to SICStus Prolog

   To translate an AML model to a SICStus Prolog model, run the script `aml2prolog` in a bash shell:

   ```
   $ ./aml2prolog inputfile
   ```

   where `inputfile` is a AML-Model. The generated Prolog file will be written into `inputfile.pl` .

3. Translating AML to and XML description

   To translate an AML model to a XML description, run the script `aml2xml` in a bash shell:

   ```
   $ ./aml2xml inputfile
   ```

   where `inputfile` is a AML-Model. The generated XML file will be written into `inputfile.xml`.

4. Translating an XML description to a Gecode-adapted C++ model

   To translate an XML description to a C++ model, run the script `xml2cc` in a bash shell:

```
$ ./xml2cc inputfile
```

where `inputfile` is a XML-description following the DTD in `examples/model.dtd`. The generated C++ model will be written into `inputfile.cc`.

5. Translating an XML description to a SICStus Prolog model

   To translate an XML description to a Prolog model, run the script `xml2prolog` in a bash shell:

   ```
   $ ./xml2prolog inputfile
   ```

   where `inputfile` is a XML-description follwing the DTD in `examples/model.dtd`. The generated Prolog model will be written into `inputfile.pl` .

Appendix B

AML Grammar

This section presents the context free grammar of the AML modeling language. Terminals are written in capital letters and the $epsilon$-edge is denoted by `/* epsilon */`

```
aml_model -> MODEL IDENT parameters LBRACE
               data_declaration
               var_declaration
               SOLVE LBRACE constraint_list RBRACE
               options
             RBRACE
           | MODEL IDENT LBRACE
               data_declaration
               var_declaration
               SOLVE LBRACE constraint_list RBRACE
               options
             RBRACE

parameters -> LPAREN parameter_list RPAREN

identifier ->   IDENT
              | IDENT LBRACK NUM RBRACK

loop_identifier -> IDENT LBRACK expression RBRACK

parameter_list -> INT IDENT
             | INT IDENT LBRACK NUM RBRACK
             | parameter_list COMMA INT IDENT
```

```
            | parameter_list COMMA INT IDENT LBRACK NUM RBRACK
            | IDENT

data_declaration ->
              DATA LBRACE const_declaration_list RBRACE
            | /* epsilon */


const_declaration_list -> const_declaration
        |  const_declaration_list const_declaration

const_declaration ->
          INT IDENT LBRACK NUM RBRACK SEMICOLON
        | INT identifier ASSOP expression SEMICOLON
        | identifier ASSOP expression SEMICOLON


var_declaration ->
       VAR INT identifier range SEMICOLON
     | VAR INT identifier SEMICOLON
     | VAR INT IDENT LBRACK expression RBRACK
         SEMICOLON
     | VAR INT IDENT LBRACK expression RBRACK
         range SEMICOLON
     | var_declaration
         VAR INT identifier SEMICOLON
     | var_declaration
         VAR INT identifier range SEMICOLON
     | var_declaration VAR INT IDENT
         LBRACK expression RBRACK SEMICOLON
     | var_declaration VAR INT IDENT
         LBRACK expression RBRACK range SEMICOLON

range -> IN expression DOTDOT expression


constraint_  ->  relation
    | DISTINCT LPAREN IDENT RPAREN SEMICOLON
    | SERIALIZED LPAREN IDENT COMMA IDENT
          RPAREN SEMICOLON
```

```
    | CUMULATIVE LPAREN IDENT COMMA IDENT COMMA
          IDENT COMMA expression RPAREN SEMICOLON
    | forloop

constraint_list -> constraint_
      | constraint_list constraint_

expression ->   NUMBER
             | identifier
             | LPAREN expression RPAREN
             | expression PLUS expression
             | expression MINUS expression
             | expression MULT expression
             | expression DIV expression

relation ->  expression EQ expression SEMICOLON
             | expression GT expression SEMICOLON
             | expression LT expression SEMICOLON
             | expression GTEQ expression SEMICOLON
             | expression LTEQ expression SEMICOLON
             | expression NOTEQ expression SEMICOLON

forloop ->  FORALL LPAREN identifier range RPAREN LBRACE
      loop_constraints_list
    RBRACE

loop_constraints_list ->   loop_constraint
     | loop_constraints_list loop_constraint

loop_expression ->  NUM
             | identifier
             | loop_identifier
             | LPAREN loop_expression RPAREN
             | loop_expression PLUS loop_expression
             | loop_expression MINUS loop_expression
             | loop_expression MULT loop_expression
             | loop_expression DIV loop_expression

loop_constraint ->
        loop_expression relop loop_expression SEMICOLON
```

```
      | DISTINCT LPAREN IDENT RPAREN SEMICOLON
      | SERIALIZED LPAREN IDENT COMMA IDENT RPAREN SEMICOLON
      | CUMULATIVE LPAREN IDENT COMMA IDENT COMMA
           IDENT COMMA expression RPAREN SEMICOLON
      | forloop

relop ->    EQ
          | GT
          | LT
          | GTEQ
          | LTEQ
          | NOTEQ

options ->   OPTIONS LBRACE options_list RBRACE
     | OPTIONS LBRACE RBRACE
     | /* epsilon */


options_list -> options_list ITERATIONS ASSOP NUM SEMICOLON
      | options_list SOLUTIONS ASSOP NUM SEMICOLON
      | options_list IDENT ASSOP expression SEMICOLON
      | options_list IDENT  SEMICOLON
      | options_list BRANCHINGVAR ASSOP IDENT SEMICOLON
      | options_list BRANCHINGVAL ASSOP IDENT SEMICOLON
      | BRANCHINGVAL ASSOP IDENT SEMICOLON
      | BRANCHINGVAR ASSOP IDENT SEMICOLON
      | ITERATIONS ASSOP NUM SEMICOLON
      | SOLUTIONS ASSOP NUM SEMICOLON
      | IDENT ASSOP expression SEMICOLON
      | IDENT  SEMICOLON
      | /* epsilon */
```

Bibliography

[1.5] The Java 2 Standard Edition Application Programming Interface. version 1.5.0. Availabe at `http://java.sun.com/j2se/1.5.0/docs/api/`.

[Ag04] M. Å gren. ”introducing esra, a relational language for modelling combinatorial problems”. *LOPSTR'03: Revised Selected Paper, LNCS 3018,214-232*, 2004.

[Apt03] K.R. Apt. *”Principles of Constraint Programming”*. Cambridge University Press, 2003.

[ASU86] A.V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[BVH03] C. Bessiere and P. Van Hentenryck. ”to be or not to be ... a global constraint”. *CP 2003 789-794*, 2003.

[Cea95] M. Carlsson et al. *”SICStus Prolog User's Manual”, Release 3*. Swedish Institute of Computer Science, Sweden, 1995.

[COC97] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver, 1997.

[FJMM05] A. Frisch, C. Jefferson, B. Martinez, and I. Miguel. ”rules of constraint modeling”. *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, UK*, 2005.

[gcde] gecode generic constraint development environment. Available from `http://www.gecode.org`.

[Hol95] C. Holzbaur. Ofai clp(q,r) manual, edition 1.3.3, 1995.

[Lau78] J-L. Lauriere. ”a language and a program for stating and solving combinatorial problems”. *Artificial Intelligence 10(1)*, pages 29–127, 1978.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *lex & yacc*, *Second Edition*. O’Reilly & Associates, Inc., 1992.

[MC] The Mozart programming system MOzart Consortium. Available from `http://www.mozart-oz.org`.

[Rém00] D. Rémy. Using, understanding, and unraveling the ocaml language. from practice to theory and vice versa. In *APPSEM*, pages 413–536, 2000.

[Sch99] C. Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, 1999. The MIT Press.

[Smo95] G. Smolka. ”the oz programming model”. In *Computer Science Today*, pages 324–343. 1995.

[Sys] The Alice System. Availabe at `www.ps.uni-sb.de/alice`.

[VH99] P. Van Hentenryck. *”The OPL Optimization Programming Language”*. Massachusetts Institute of Technology, Cambridge, 1999.