Chapter 4

THEORETICAL CONCEPTS AND DESIGN OF MODELING LANGUAGES FOR MATHEMATICAL OPTIMIZATION

Hermann Schichl[*]
Institut für Mathematik der Universität Wien
Strudlhofgasse 4, A-1090 Wien
Hermann.Schichl@esi.ac.at

Abstract In this chapter we will present the basic design features of modeling languages, turning our attention to algebraic modeling languages. Later we will introduce an important class of optimization problems — global optimization, and illustrate the difficulties in constructing models for such problems.

Keywords: Modeling, Modeling Language, Modeling System, Modeling Software, Algebraic Modeling Language, Declarative Language, Global Optimization

4.1 Modeling Languages

The development of modeling languages started in the late 1970s when `GAMS` was designed, although, as described in Section 1.3 there had been some precursors before.

Since that time all the error prone routine tasks in Fig. 2.4.3 could be performed by the computer. The process of modeling became much more convenient, and the flexibility was increased a lot.

[*]funded by EU project COCONUT IST-2000-26063

In a modeling language, the model can be written in a form which is close to the mathematical notation, actually a typical feature of an *algebraic modeling language* (see Section 4.1.1).

The formulation of the model is *independent of solver formats*. Different *solvers* can be connected to the modeling language, and the translation of models and data to the solver format is done automatically. This has several advantages. The formerly tedious and error prone translation steps are done by the computer, and after thorough testing of the interface errors are very unlikely. There is a clean cut between the modeling and the numerical, algorithmic part. In addition, for hard problems *different* solvers can be tried, making it more likely that a solution algorithm is found which produces a useful result.

In a modeling language, model and model data are kept separately. There is a clear cut between the model structure and the data. Thus, many different instances of the same model class with varying data can be solved. Many systems provide an ODBC (open database connectivity) interface for automatic database access and an interface to the most widely used spreadsheet systems. This relieves the user from the laborious duty of searching for the relevant data every time the model is used. A second advantage of this concept is that during the development phase of the model (in the cycle) the approach can be tested on *toy problems* with small artificial data sets, and later the model can be applied without change for large scale industry-relevant instances with real data.

Even the problem of *derived data* is solved in most modeling systems. Due to the development of *automatic differentiation* (see *e.g.*, [88]) the modeling languages can generate derived information (gradients, sparse Hessians,. . .) from the model description without assistance of the user.

Today, there are several modeling systems in use, as described in Section 1.3. At first we will analyze the biggest class, the algebraic modeling languages, in Section 4.1.1. In the later sections we will shed light on the other language classes, ending with languages.

4.1.1 Algebraic Modeling Languages

This is the biggest class of modeling languages. Typical representatives are `GAMS` [28] (see Chapter 8), `AMPL` [72] (Chapter 7), `Xpress-MP` [5], `LINGO` [195] (Chapter 9), `NOP` [165], `NOP-2` [189] (Chapter 15), `Numerica` [217], and `MINOPT` [199] (Chapter 11).

In [106] and [104] TONY HÜRLIMANN describes why algebraic modeling languages can be viewed as a new paradigm of programming. Usually programming languages are divided into three different classes:

Imperative Languages These languages are also called *procedural languages*, typical representatives are `C`, `C++`, `Pascal`, `FORTRAN`, `Java`. With these languages a computer is programmed in a way which has a strong connection to the von Neumann concept of a computer. The stage of the computation can always be described by the state of the computer's memory locations, its instruction pointer, and some status registers. At each step this state changes until the computation reaches a terminal state, usually when the desired result is computed. The ability to assign values to variables and change the corresponding memory location in that way, and the explicit sequential execution are characteristica of an imperative language. An extension to that concept, as HÜRLIMANN expresses it: the ultimate consequence, is *object oriented programming* where computation proceeds by changing the local state of objects.

Functional Languages This paradigm is based on the concept that every computation can be viewed as a function $f : X \to Y$ translating an input from X to a unique output in Y. Since every value can be represented as a function, no variables are necessary, and there is no need for an assignment operation. There is a distinction between function definition and application of a function, and most important, functions are viewed as values themselves. A typical representative of functional programming languages is `LISP`.

Logic Programming Languages This paradigm was developed in the 1960s during the construction of theorem provers, programs which were designed to prove mathematical theorems. In this development process it was detected that every mathematical proof can be regarded as a computation following specific rules. In the reverse, every computation can as well be regarded as a proof, and so the most important representative of this language category, the programming language `Prolog` was designed.

Every program in `Prolog` consists of a non-empty set of goals and a set of *(Horn)-rules*[1]. Using the backtracking algorithm, or more modern resolution methods (*constraint logic programming*, see [113]) the program tries to reach a goal obeying the rules. As a short summary, one could say that a logic programming language can be seen as a "notational system for writing logical statements together with specified algorithms for implementing inference rules" [144, p. 426]. Typical logic programming languages are `Prolog V` and `ECLiPSe`.

All languages from these classes specify a problem in an algorithmic way. Like in Babylonian mathematics, it is not specified *what the problem is* but rather

[1]A *Horn-rule* or *Horn-clause*is a disjunction of literals (i.e. atomic statement) with at most one positive literal. If the clause contains only the positive literal, it is called a *fact*.

how to solve the problem. For this reason we call these languages *algorithmic languages*.

In contrast to that, modeling languages store the *knowledge* about a model, they *define the problem* and usually do not specify how to solve it. They are *declarative languages*.

The important question here is: How do we describe what the problem is? The answer is: By specifying the problem's properties.

We start with a state space X (usually $\mathbb{R}^n \times \mathbb{Z}^m$). On this state space we define a set of *constraints* by mathematical formulas (usually *equations*, *inequalities*, and *optimality requirements*), which together define a relation $R : X \to \{\text{true}, \text{false}\}$ deciding about the *admissibility* of a member $x \in X$. We say x is *admissible*[2] if no constraints are violated, *i.e.*, if $R(x) = \text{true}$, and we define the *mathematical model* $M := \{x \in X \mid R(x)\}$ for the problem we are considering. This formulation is very close to the modern mathematical approach, and it has an important consequence:

Since no solution algorithm is specified, there is a priori no reason, why a problem defined in the declarative way should be solvable at all, and there is definitely no reason why there should be an algorithm for finding a solution.

Fortunately, however, there are large classes of problems (like *linear programming*) for which algorithms to explicitly find the set M exist, and the number of solution algorithms for various problem categories grows steadily.

If we direct our attention back to declarative languages, we see their three most important aspects:

Problems are represented in a declarative way.
There is a clear separation between problem definition and the solution process.
There is a clear separation between the problem structure and its data.

Algebraic modeling languages are a special class of declarative languages, and most of them are designed for specifying *optimization problems*. Usually they are capable of describing problems of the form

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & F(x) = 0 \\ & G(x) \leq 0 \\ & x \in \mathbf{x}, \end{array} \tag{4.1.1}$$

or something slightly more general. Here $\mathbf{x}$ denotes a subbox of $X = \mathbb{R}^m \times \mathbb{Z}^n$.

[2] I have chosen the word *admissible*, since in optimization the word *feasible* is used to describe all points which obey the equality- and inequality constraints but not necessarily the optimality requirement.

The problem is flattened, i.e. all variables and constraints become essentially one-dimensional, and the model is written in an *index-based* formulation, using *algebraic expressions* in a way which is close to the mathematical notation. Typically, the problem is declared using *sets*, *indices*, *parameters*, and *variables*.

Conceptually similar entities are grouped in sets. The entities in the sets are later referenced by indices to the elements of those sets. Groups of entities (variables, constraints) can then be compactly represented and used in algebraic expressions. Here it must be noted that one of the most important characteristica of modern algebraic modeling languages is their ability to describe *non-linear models*.

In `AMPL`, for instance, the expression

$$\sum_{i \in S} x_i$$

would be written as

```
sum { i in S } x[i];
```

This leads to a problem formulation which is very close to a mathematical formulation, and translation of mathematical models to declarations in a modeling language usually only involves syntactic issues.

The algebraic modeling language is then responsible for creating a *problem instance* that a solution algorithm can work on. This is done by expanding the compact notation by indexing all the sets and adding the model data; this is often called the *set indexing ability* of algebraic modeling languages. Very often the modeling language provides an additional *presolve* phase before transferring the problem to the solution algorithm. In this phase the model is analyzed and certain simplification techniques are applied. This flattened and simplified model is then passed on to the solver.

As a small example we consider the problem

$$\begin{aligned} \text{min} \quad & x^T Q x + c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & \|x\| \geq 1 \\ & x \in [l, u], \end{aligned} \tag{4.1.2}$$

with an $N \times N$-matrix Q and an $M \times N$-matrix A.

Writing down all matrix products in index form we transform problem (4.1.2) into the "flat" model

$$\begin{aligned}
\min \quad & \sum_{i=1}^{N}\left(\sum_{j=1}^{N} Q_{ij}x_j + c_i\right)x_i \\
\text{s.t.} \quad & \sum_{j=1}^{N} A_{ij}x_j \le b_i \quad \forall i = 1,\dots,M \\
& \sqrt{\sum_{i=1}^{N} x_i^2} \ge 1 \\
& x_j \in [l_j, u_j] \quad \forall j = 1,\dots,N.
\end{aligned} \tag{4.1.3}$$

This model can then easily be "programmed" in *e.g.*, `AMPL`:

```
### PARAMETERS ###
param N>0 integer;
param M>0 integer;
param c {1..N};
param b {1..M};
param Q {1..N,1..N};
param A {1..M,1..N};
param l {1..N};
param u {1..N};
### VARIABLES ###
var x {1..N};

### OBJECTIVE ###
minimize goal_function:
  sum {i in 1..N} (sum {j in 1..N} Q[i,j]*x[j] + c[i]) * x[i];

### CONSTRAINTS ###
subject to linear_constraints {j in 1..M}:
   sum {i in 1..N} A[j,i]*x[i] <= b[j];
norm_constraint:  sqrt(sum {j in 1..N} x[j]^2) >= 1;
box_constraints {j in 1..N}: l[j]<=x[j]<=u[j];

################ DATA ####################
data sample.dat;

###########################################
solve; display x;
```

By reading through the definition, we can easily identify the various parts of the flat model, only complemented by declarations of parameters and variables. In the data section we define that the model data should be read from the file `sample.dat`, and the last line contains the only *procedural statements*, `solve` which calls the solver, and `display x` which prints the solution.

Before we consider different approaches to modeling languages, here is a short summary on the most important design features of algebraic modeling languages:

Variables, constraints with arbitrary names,
Sets, indices, algebraic expressions (possibly non-linear),
Notation which is close to the mathematical formulation,
Problem structure is data independent,
Models can be scalable, by a change of the parameters a switch can be made from *toy-problems* to *real-life problems*.
Most statements are declarative, except for *conditionals*, *loops*, and very few procedural statements,
Flexibility in the types of models that can be specified,
Convenient for the modeler (little overhead in model formulation. . .),
Simple interface between modeling language and solver

- It must be easy to connect a solver to the modeling language and to have easy access to the model, the data, and the derived information like derivatives.
- Sending data back from the solver to the modeling language like progress reports, solutions, or error messages should be straightforward.

Simple and powerful data handling (ODBC, spreadsheet interfaces, data files,. . .),
Automatic differentiation.

For a further analysis see [75].

4.1.2 Non-algebraic Modeling Languages

In certain areas, algebraic modeling languages have not been very successful.

Sometimes, writing a model with variables and constraints is just to tedious, especially if the model is highly structured.

For chemical engineering there are two modeling systems `gPROMS` [6] and `ASCEND` [170] which build models in an object oriented way. For manufacturing processes a system called `EXTEND` [180] is available. All models are built from primitive entities (*e.g.*, reactors or tanks), and these can be assembled in compound entities with certain attributes. The modeler only needs to specify the parts and their connections, and the modeling system builds the mathematical model automatically. It introduces variables and derives constraints from physical and chemical principles like the law of mass conservation.

A very recent development combining these ideas came from the ESPRIT project “Simulation in Europe Basic Research Working Group (SiE-WG)” led by a group of simulation experts. The combined effort of the members of this project led to the development of the new *object-oriented modeling language* Modelica [56].

In *constraint logic programming* there is a different reason for the lack of interest in algebraic modeling languages. For most of the difficult discrete problems there is no generic algorithm available which can solve the problems without hints from the modeler.

Hence, for solving these hard problems, it is necessary to store at least partial algorithmic knowledge together with the problem definition. Most of the algebraic modeling languages do not provide this possibility. Even more, discrete problems usually need model descriptions with very specialized constraints (all_diff — in a set of variables all take different values, cardinality — exactly N of a given set of variables have the value *true*). These discrete constraints are usually not provided by algebraic modeling languages. There they have to be reformulated using special mixed-integer linear constraints (see [123, Chap. 5]), which sometimes is inefficient and hides the model structure.

In these areas the modern constraint logic programming languages like ECLiPSe are usually used. A sample file is given below. It can be used to solve the famous n-queens problem using various search and backtracking strategies.

```
% ECLiPSe SAMPLE CODE
%
% AUTHOR:       Joachim Schimpf, IC-Parc
%
% The famous N-queens problem, using finite domains
% and a selection fo different search strategies
%
:- set_flag(gc_interval, 100000000).
:- lib(lists).
:- lib(fd).
:- lib(fd_search).
%--------------------
% The model
%--------------------
queens(N, Board) :-
    length(Board, N),
    Board :: 1..N,
    ( fromto(Board, [Q1|Cols], Cols, []) do
        ( foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
            noattack(Q1, Q2, Dist)
        )
    ).

noattack(Q1,Q2,Dist) :-
    Q2 #\= Q1,
    Q2 - Q1 #\= Dist,
    Q1 - Q2 #\= Dist.
%----------------------
% The search strategies
%----------------------
labeling(a, AllVars) :-
    ( foreach(Var, AllVars) do
        count_backtracks,
        indomain(Var)              % select value
    ).

% functions labeling(b,.. to labeling(d,.. deleted here !

labeling(e, AllVars) :-
```

```
\   middle_first(AllVars, AllVarsPreOrdered), % static var-select
    ( fromto(AllVarsPreOrdered, Vars, VarsRem, []) do
%      search_space(Vars, Size), writeln(Size),

\       delete(Var, Vars, VarsRem, 0, first_fail),   % dynamic var-select
        count_backtracks,
        indomain(Var, middle)               % select value
    ).

% reorder a list so that the middle elements are first
middle_first(List, Ordered) :-
         halve(List, Front, Back),
         reverse(Front, RevFront),
         splice(Back, RevFront, Ordered).
%----------------------------------
% Toplevel code
%
% all_queens/2 finds all solutions
% first_queens/2 finds one solution
% Strategy is a,b,c,d or e
% N is the size of the board
%----------------------------------
all_queens(Strategy, N) :-             % Find all solutions
    setval(solutions, 0),
    statistics(times, [T0|_]),
    (
       queens(N, Board),
       init_backtracks,
       labeling(Strategy, Board),
%     writeln(Board),
%     put(0'.),
       incval(solutions),
       fail
    ;
       true
    ),
    get_backtracks(B),
    statistics(times, [T1|_]),
    T is T1-T0,
    getval(solutions, S),

\   printf("\nFound %d solutions for %d queens in %w s with %d backtracks%n",
           [S,N,T,B]).

first_queens(Strategy, N) :-           % Find one solution
    statistics(times, [T0|_]),
    queens(N, Board),
    statistics(times, [T1|_]),
    D1 is T1-T0,

\   printf("Setup for %d queens done in %w seconds", [N,D1]), nl,
    init_backtracks,
    labeling(Strategy, Board),
    get_backtracks(B),
    statistics(times, [T2|_]),
    D2 is T2-T1,

\   printf("Found first solution for %d queens in %w s with %d backtracks%n",
           [N,D2,B]).

%--------------------
% Utilities
%--------------------
search_space(Vars, Size) :-
    ( foreach(V,Vars), fromto(1,S0,S1,Size) do
         dvar_domain(V,D),
         S1 is S0*dom_size(D)
    ).
```

```
:- local variable(backtracks), variable(deep_fail).
init_backtracks :-
        setval(backtracks,0).
get_backtracks(B) :-
        getval(backtracks,B).
count_backtracks :-
        setval(deep_fail,false).
count_backtracks :-
        getval(deep_fail,false),        % may fail
        setval(deep_fail,true),
        incval(backtracks),
        fail.
```

Looking at this file two facts become immediately obvious. First, it is big although the model description needs only about 15 lines near the beginning. Second, the declarative information is only a very small part of the model. The remaining file contains a lot of algorithmic knowledge: two solution strategies, one for searching all solutions and one for finding only one.

Finally, there are modeling languages which directly interface to spreadsheet systems. This is especially convenient for users who do not want to bother about learning new systems. They can enter their data into the spreadsheets as usual, and afterwards they add a model by specifying spreadsheet formulas and bounds for certain cells. Then an add-on program for the spreadsheet system solves the so-entered model and stores the result back into the “variable cells”. For MS Excel there is a *spreadsheet modeling system* by FRONTLINE Systems.

4.1.3 Integrated Modeling Environments

A more graphical approach to model building is taken by the so called integrated modeling environments.

They do not primarily contain a language for model building with interfaces to different solvers but rely on *graphical user interfaces* (GUI). Usually, the models are represented by a database, and model building is done in a menu-driven way.

In addition to a model building facility these systems also provide *GUI-builders* for end-user applications. State of the art *visualization techniques* give insight into the solving process and can be used to illustrate the solutions.

Two very prominent representatives of this approach are AIMMS (see Chapter 6) by Paragon Systems, MPL by Maximal Software, and OPL Studio (see 17) by ILOG.

4.1.4 Model-Programming Languages

Recently, several languages try to bridge the gap between the algorithmic languages used in constraint programming and the declarative languages mainly used in mathematical optimization.

Many modeling languages nowadays have built-in scripting capabilities (see Chapters 7 and 8), but these new languages provide elements for declaring models and for describing solution algorithms in a much more integrated way.

The `Mosel` [96] (Chapter 12) approach overall looks much like a programming language, even the model declaration part has to be programmed.

The language `LPL` [100] (Chapter 10), on the other hand, provides a declarative part which still looks similar to mathematical notation, and the algorithmic part has the look and feel of an imperative programming language.

Also `AIMMS` (Chapter 6) can be seen as a modeling system which combines declarative and algorithmic elements. There, however, the GUI is the main focus, and declarations and programming are based on it.

Basically, a trend can be seen that modeling languages provide the possibility to record algorithmic knowledge in addition to purely declarative modeling. This can be an advantage for the modeler because of increased flexibility, but the future will show whether this approach makes models more or less maintainable, and whether it increases or decreases the shelf-life time of the models.

The approach that the modeling system provides just the models and that the solver interface contains the algorithms is still widely used. This has the advantage that different solution algorithms can get different algorithmic “hints”. However, it is more difficult to document the whole model solution process with this approach, especially if we consider that, *e.g.*, `LPL` provides automatic document generation capabilities.

4.1.5 Other Modeling Tools

Apart from modeling systems and modeling languages there are tools for analyzing already existing models.

The `ANALYZE` system [85] provides various tools for linear programming problems. An LP can examined w.r.t. *constraint effectiveness*, *objective effect*, and *grouping of constraints* (the system can also make suggestions on better constraint sorting).
For non-linear models there is a successor product. `MPROBE` [36] provides in addition to the aforementioned methods also a shape analysis tool for non-linear model parts.

4.2 Global Optimization

It was already outlined in Section 1.3 that the development of modeling languages started with the invention of matrix generators. At that time, linear programs were the only real-life models which could be effectively solved.

As the computing power increased and the solution algorithms became more effective, higher and higher dimensional nonlinear programs were made accessible. The modeling systems had to keep up with the development, and ultimately this was one reason for the emergence of algebraic modeling languages. The algorithms available solved *local optimization* problems in an approximate way. So most of the now available modeling systems provide very good connectivity for linear solvers and local optimization algorithms.

Modeling and computer aided model analysis can serve many different needs. As was pointed out in [123] various levels of rigor can be distinguished for models, depending on the modeling goal.

$$\text{Validation} \leq \text{Verification/Falsification} \leq \text{Mathematical Proof}$$

The highest level of rigor is necessary for *mathematical proofs*. There are real-world applications which need a similar level of rigor, *e.g.*, in chemical engineering and robotics. Most of these problems are global optimization problems. They are of the form

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & F(x) \in \mathbf{F} \\ & x \in \mathbf{x}, \end{array} \tag{4.2.4}$$

where $\mathbf{F}$ and $\mathbf{x} \subseteq X$ are both boxes (multidimensional intervals), and $X = \mathbb{R}^n \times \mathbb{Z}^m$. In contrast to local optimization, the task is to find *all globally best* solutions to this problems and to do it in a rigorous way, *i.e.*, to compute the solution set S of (4.2.4) and a mathematical proof that S is the solution set.

When solving such models harmful approximations must not be performed, and a lot of techniques have to be applied which are not usually necessary for local optimization. The most important tool is *interval arithmetic* with *directed rounding*.

During the last few years an increasing number of algorithms were developed for global optimization (*e.g.*, BARON [185], αBB [3], GLOBSOL [128], and the COCONUT project [40]). Some of the solvers are now reaching a strength which enables them to solve some industry-relevant problems.

In this section we will glance at global optimization and pinpoint the difficulties which arise when interfacing algebraic modeling languages with global optimization software.

4.2.1 Problem Description

Let us consider again problem (4.2.4). Here we describe the most general problem class GLOBAL (see Section 1.2.1). The task is to find *all* $x \in \mathbf{x}$ with $F(x) \in \mathbf{F}$ and the property that

$$\forall y \in \mathbf{x} \text{ with } F(y) \in \mathbf{F} : \ f(y) \geq f(x).$$

The variables x can come in different flavours

continuous:	The variable is real or complex valued, and usually bounds are provided for its domain (*e.g.*, the amount of fluid flowing through a pipe in a chemical plant).
integer:	These are discrete variables which can take all integers (or complex integers) as values. Sometimes they are in addition bound constrained (*e.g.*, the number of television sets produced by some production site).
binary:	The value is either 0 or 1. These variables are also called *decision variables* (*e.g.*, decide whether or not a certain chemical reactor is to be included when building a new plant).
discrete variables:	These can take a finite number of values which do not need to be numbers.
semi-continuous:	They are either zero or bigger as some threshold (*e.g.*, we either do not start production in facility A at all or we have to manufacture a minimum amount of goods).
partial integer:	These variables take either integer values in $0, \dots, N$ or continuous values in $[N, M]$.

There are several types of constraints F

continuous:	These constraints mainly involve continuous variables. Usually they are piecewise differentiable and composed from elementary functions. They can be linear, quadratic, convex, or nonconvex.
discrete:	They only take discrete values. Classification constraints are of this type.
logical:	Constraints of this type involve logical operators and binary variables.
global:	The number of variables involved in a single constraint is small for most real-life models. There are, however, some constraints which involve all or just a lot of variables. These are called global constraints. Typical representatives are *all-different*, *cardinality*[3], and *histogram*[4] constraints.
special ordered sets:	there are two categories: **SOS-1** At most one from a set of variables is non-zero, all the others vanish. **SOS-2** At most two variables from a given ordered list of variables are non-zero, and if two are non-zero, they have to be neighbors in the list.
exclusion regions:	these are constraints of the form $x \notin \mathbf{x}^e$.

Compared to models with low requirements in rigor, for global optimization much more care has to be taken in data handling. For mathematical proofs all *data* items must be *known exactly*. *Round-off* in the translation of the input data from modeling system to the solver can destroy important model properties. E.g. the very simple system

$$\begin{aligned} 0.4x + 0.6y &\geq 1 \\ x, y &\in [0, 1] \end{aligned} \tag{4.2.5}$$

has exactly one solution $x = y = 1$. The system becomes infeasible when rounded to IEEE floating point representation.

For some problems the *data* might be *uncertain*, see Section 1.2.7. For example the elasticity module for steel bars can vary by more than 15%. For rigorous problem solving it is important that all parameters can be entered together with their amount of uncertainty.

Sometimes constraints are less relevant than others or they may be slightly violated in some circumstances. E.g. the radiation patterns of radio therapy for cancerous tumors can be optimized. There are constraints on the minimal dose needed for the tumor cells, and maximum doses for the remaining tissue types. Sometimes there is no solution, depending on the size of the tumor and the geometric shapes. Then the physician might say: “Well, I don’t mind if the liver gets a higher dose than allowed, as long as this excess radiation is confined to a small part of the tissue.” In these cases we call the constraint a *soft constraint*.

4.2.2 Algebraic Modeling Languages and Global Optimization

Most modeling systems around have originally been developed for nonlinear local optimization and linear programming. When trying to connect a solver for global optimization problems to such a modeling language, the programmer usually runs into one or all of the following problems. Therefore, the global optimization community has developed its own input languages or modeling languages, like `Numerica` [217], `NOP` [165], or `NOP-2` [189] (see Chapter 15).

The difficulties start with the fact that most modeling systems pass a presolved problem, and all input data is rounded to floating point representation. Sometimes it is not even possible to decide whether a datum the solver sees is integer or only some real number very close to an integer. This involves round-off and a loss of important knowledge and makes mathematical rigor impossible.

The second point is that the performance of global optimization algorithms usually depends on whether specific structural knowledge about the problem can be exploited. Unfortunately, the problem which is passed from the modeling system to the solver is “flat” and has lost most of its mathematical structure, except for the sparsity pattern.

There is almost no modeling language around in which uncertain data can be specified. This makes it, *e.g.*, almost impossible to model safety programs (see Section 1.2.6).

There is only restricted support for soft constraints.

Many modeling languages only support the first three variable categories (continuous, integer, binary), and there is almost no support for global constraints (all different, cardinality, histogram,. . .) and exclusion regions.

Automatic differentiation is not sufficient for constructing all derived information needed for a global optimization algorithm. There is no inherent support for *interval analysis*, and hence essential data such as implied bounds cannot be generated automatically (except in Numerica).

There is no support for matrix operations, so all matrix operations have to be flattened, hence the structure is lost (consider, *e.g.*, the constraint $A^T DAx = b$). Some matrix operations cannot be used inside a model at all, like $\det(A) \geq c$, or A is *positive semidefinite*, specifying these constraints is impossible in all modeling languages I know.

Finally, many global optimization problems can only be solved by using several *mathematically equivalent formulations* of the same model at the same time. A very nice example for that principle from an application in robotics can be found in [138]. There is no modeling language which supports equivalent formulations.

4.3 A Vision — What the Future Needs to Bring

In this section we will make a few technical suggestions about extensions to the existing modeling languages which are necessary in order to be able to write global and stochastic optimization problems and multi-stage problems in a convenient way.

4.3.1 Data Handling

From the declarative part, which specifies the model structure, the modeling system generates the *problem instance* by adding the model data. This principle was already described in Section 4.1.1.

Since data is a very important part in the modeling process and later during the use of the model, it is important that knowledge about the data is recordable as well.

- The *source of the data* is important, and how much it suits the model.
- *Accuracy* or estimates on data accuracy should be recordable in the modeling system.
- Directed rounding, interval enclosures, and exact values should be supported for rigorous computing (see Chapter 15).
- The date of data construction is probably interesting, as well, and how long it can be considered *up-to-date*.
- Data *validity* and *completeness* is also relevant.

Data handling should be *convenient*. An appropriate interface to automatic data retrieval and data management tools should be included in a modeling system. There should be convenient ways to access *databases* (ODBC) and *spreadsheet* data.

4.3.2 Solver Views

Constraint Logic Programming (CPL) and their languages teach that for the processing of very complex models the declarative knowledge usually is not sufficient for solving the problem, as explained in Section 4.1.4.

Depending on the solver used, additional hints must be specified to guide the algorithm. These hints are neither part of the model structure nor part of the model's data. Different solvers will need different hints and guides for handling the same model instance. Useful additional information includes *search and selection strategies*, *scenarios for stochastic programs*, *starting points for local optimization*, *convex underestimating functions*, *valid cuts*, *bisection and backtracking hints*, *switching between mathematically equivalent model formulations*.
These hints can be kept separate for every solver used, and the modeling system would be responsible to construct a *solver's view of the model instance*, which is a mixture of declarative and algorithmic knowledge.

4.3.3 GUI

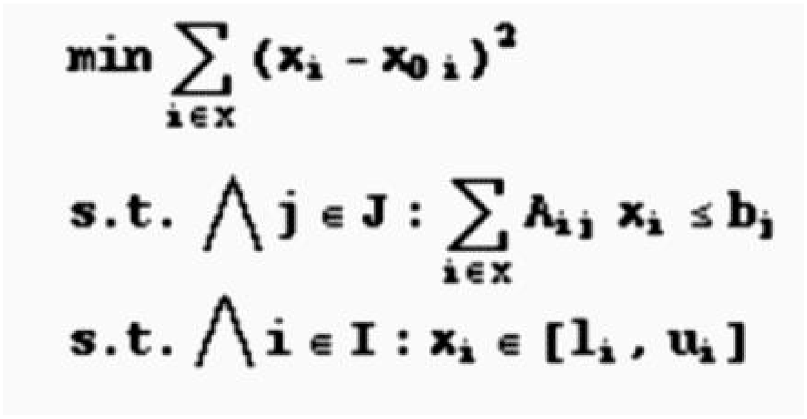
The introduction of GUI front-ends, which is, *e.g.*, done in `AIMMS` (see Chapter 6) and `OPL Studio` (see Chapter 17) provides the modeler with additional possibilities.

- With graphical capabilities and additional fonts, there could be an even *more mathematical notation*. It looks similar to the front-end of Mathematica 4, like in Fig. 4.3.1,
- Provide *model structuring* with coarse and fine grained views.
- Offer *syntax based editing* capabilities with *online help*.
- Visualize the connection between the model parts, cross referencing between variables and constraints.
- *Visualize Progress* during the solution process.
- *Visualize Result*.

4.3.4 Object Oriented Modeling — Derived Models

In the same way that object oriented programming is the ultimate consequence of imperative programming (*cf.* Section 4.1.1), the declarative languages need the same kind of extension: *derived models*.

This mimics the way mathematicians define new objects. The algebraic structure of a group is defined to be a monoid in which every element has an inverse. The remaining properties like the law of associativity and the existence of a unit element are "inherited" from the monoid structure. As a further example, we take the definition of an algebra as a vector space, which is also a ring with the same additive group, and additional compatibility laws are valid.



$$\min \sum_{i \in X} (x_i - x_{0\,i})^2$$

$$\text{s.t.} \bigwedge j \in J : \sum_{i \in X} A_{ij}\, x_i \leq b_j$$

$$\text{s.t.} \bigwedge i \in I : x_i \in [l_i, u_i]$$

Figure 4.3.1. Model in a GUI front-end

The same could be done in modeling. A derived model would be a model which has all properties of all the models it is an epigone of plus additional variables and constraints.

For example, a *screw* could be represented by variables and constraints and a few parameters like its size. *Steel* is a material, and all its properties are recorded with variables and constraints, and in the same way a model for *wood* is produced.

The derived modeling method now would make it straightforward to construct models for steel screws and for wooden screws.

4.3.5 Hierarchical Modeling

As described in Section 1.2.4, multi-stage models often arise naturally. For instance, in the first step we just want a method of proving that a given robot design is free of catastrophic singularities. We build a model A for that and solve it. The strict separation of model structure and data enables us to prove design correctness for many robots, and at some stage we decide that we want to compute an optimal design which is cheap and free of structural deficiencies.

The resulting problem is a two-stage design problem, where the first stage consists of model A, the only difference is that the design parameters become free variables instead of being fixed by the model data.

For making such a hierarchical approach to modeling possible, all *models* in a modeling language *should be reusable as "functions"* in bigger models.

4.3.6 Building Blocks

Many huge optimization problems are built from a number of smaller (almost) identical building blocks, especially in *chemical engineering*, *manufacturing processes*, *resource optimization*, *optimal scheduling*,. . .

Adding support for building blocks (with configurable automatic constraints when connected together) and enhancing the building block management by providing GUI support will *decrease the number of modeling errors* for big problems and *reduce the modeling costs*.

A good example for such a system is, *e.g.*, `BoFit` by `ProComm` [173], a system for the daily resource planning for electrical energy providers.

4.3.7 Open Model Exchange Format

It would be great if all modeling system providers could agree upon a common exchange interface for mathematical models. Until now it is a tedious task to translate a model from one modeling system to another.

The Open Model Exchange Format could be XML- and MathML-based. This would have the advantage that it would provide a convenient interface for model presentation on the Worldwide Web, it would need a consistent interface for floating point numbers which does not produce read/write mismatches due to rounding errors, and it should be easily extendable.

It would be useful to have ASCII and binary interfaces and encryption to protect intellectual properties. The most important feature would be that it is *open*, *public*, and *widely supported*.

Acknowledgments

I want to thank Arnold Neumaier for his help and his important advice for preparing this section, and Oleg Shcherbina for his support and the many references he provided.