MODELING LANGUAGES IN OPTIMIZATION, *a new paradigm*

In this paper, modeling languages are identified as a new computer language paradigm and their applications for representing optimization problems is illustrated by examples.

Programming languages can be classified into three paradigms: imperative, functional, and logic programming [14]. The *imperative programming paradigm* is closely related to the physical way of how (the von Neumann) computer works: Given a set of memory locations, a program is a sequence of well defined instructions on retrieving, storing and transforming the content of these locations. The *functional paradigm* of computation is based on the evaluation of functions. Every program can be viewed as a function which translates an input into a unique output. Functions are first-class values, that is, they must be viewed as values themselves. The computational model is based on the λ–calculus invented by Church A. (1936) as a mathematical formalism for expressing the concept of a computation. The *paradigm of logic programming* is based on the insight that a computation can be viewed as a kind of (constructive) proof. Hence, a program is a notation for writing logical statements together with specified algorithms for implementing inference rules.

All three programming paradigms concentrate on problem representation as a *computation*, that is, the problem is stated in a way that describes the process of solving it. The computation on how to solve a problem *is* its representation. One may call such a notational system an *algorithmic language*.

Definition: An *algorithmic language* describes (explicitly or implicitly) the computation of solving a problem, that is, *how* a problem can be processed using a machine. The computation consists of a sequence of well-defined instructions which can be executed in a finite time by a Turing machine. The information of a problem which is captured by an algorithmic language is called *algorithmic knowledge* of the problem.

algorithmic language
declarative language

Algorithmic knowledge to describe a problem is very common in our everyday life — one only need to look at cookery-books, or technical maintenance manuals — that one may ask whether the human brain is "predisposed" to preferably present a problem in describing its solution recipe.

However, there exists at least one different way to capture knowledge about a problem; it is the method which describes *what* the problem is by defining its properties, rather than saying *how* to solve it. Mathematically, this can be expressed by a set $\{x \in X \mid R(x)\}$, where X is a continuous or discrete state space and $R(x)$ is a Boolean relation, defining the properties or the *constraints* of the problem; x is called the *variable(s)*. A notational system that represents a problem in this way is called a *declarative language*.

Definition: An *declarative language* describes the problem as a set using mathematical variables and constraints defined over a given state space. This space can be finite or infinite, countable or noncountable. The information of a problem which is captured by a declarative language is called *declarative knowledge* of the problem.

The declarative representation, in general, does not give any indication on how to solve the problem. It only states what the problem is. Of course, there exists a trivial algorithm to solve a declaratively stated problem, which is to enumerate the state space and to check whether a given $x \in X$ violates the constraint $R(x)$. The algorithm breaks down, however, whenever the state space is infinite. But even if the state space is finite, it is – for most nontrivial problems – so large that a full enumeration is practically impossible.

Algorithmic and declarative representations are two fundamentally different kinds of modeling and representing knowledge. Declarative knowledge answers the question "what is?", whereas algorithmic knowledge asks "how to?" [4]. An algorithm gives an exact recipe of how to solve a problem. A mathematical model, i.e. its declarative representation, on the other hand,

(only) defines the problem as a subspace of the state space. No algorithm is given to find all or a single element of the feasible subspace.

Why declarative representation. The question arises, therefore, why to present a problem using a declarative way, since one must solve it anyway and, hence, represent as an algorithm? The reasons are, first of all, *conciseness*, *insight*, and *documentation*. Many problems can be represented declaratively in a very concise way, while the representation of their computation is long and complex. Concise writings favour also the insight of a problem. Furthermore, in many scientific papers a problem is stated in a declarative way using mathematical equations and inequalities for documentational purposes. This gives a clear statement of the problem and is an efficient way to communicate it to other scientists. However, documentation is by no means limited to human beings. One can imagine declarative languages implemented on a computer like algorithmic languages, which are parsed and interpreted by a compiler. In this way, an interpretative system can *analyse the structure* of a declarative program, can *pretty-print* it on a printer or a screen, can *classify* it, or *symbolically transform* it in order to view it as a diagram or in another textual form.

Of course, the most interesting question is whether the declarative way of representing a problem could be of any help in *solving* the problem.

Indeed, for certain classes of problems the computation can be obtained directly from a declarative formulation. This is true for all recursive definitions. A classical example is the algorithm of Euclid to find the greatest common divisor (gcd) of two integers. One can proof that

$$\gcd(a,b) = \begin{cases} \gcd(b, a \bmod b), & b > 0 \\ a, & b = 0 \end{cases}$$

which is clearly a declarative statement of the problem. In Scheme, a functional language, this formula can be implemented directly as a function in the following way:

```
(define (gcd a b)
  (if (= b 0) a
    (gcd b (remainder a b))))
```

Similar formulations can be given for any other language which includes recursion as a basic control structure. This class of problems is surprisingly rich. The whole paradigm of dynamic programming can be subsumed under this class.

A class of problems of a very different kind are linear programs, which can be represented declaratively in the following way:

$$\{\min cx \mid Ax \geq b\}$$

From this formulation – in contrast to the class of recursive definitions – nothing can be deduced that would be useful in solving the problem. However, there exists well-known methods, for example the Simplex method, which solves almost all instances in a very efficient way. Hence, to make the declarative formulation of a linear program useful for solving it, one only needs to translate it into a form, the Simplex algorithm accepts as input. The translation from the declarative formulation $\{\min\ cx \mid Ax \geq b\}$ to such an input-form can be automated. This concept can be extended to non-linear and discrete problems.

Algebraic modeling languages. The idea to state the mathematical problem in a declarative way and to translate it into an "algorithmic" form by a standard procedure led to a new language paradigm emerged basically in the community of operations research during the last ten years, the *algebraic modeling languages* (AIMMS [1], AMPL [7], GAMS [2], LINGO [18], and LPL [12] and others). These languages are becoming increasingly popular even outside the community of operations research. Algebraic modeling languages represent a problem in a purely declarative way, although most of them include computational facilities to manipulate the data as well as certain control structures.

One of their strength is the complete separation of the problem formulation as a declarative model from finding a solution, which is supposed to be computed by an external program called a *solver*. This allows the modeler not only to separate the two main tasks of model formulation

solver

and model solution, but also to switch easily between several solvers. This is an invaluable benefit for many difficult problems, since it is not uncommon that a model instance can be solved using one method, and another instance is solvable only using another method. Another advantage of such languages is to separate clearly between model structure, which only contains parameters (place-holder for data) but no data, and model instance, in which the parameters are replaced by a specific data set. This leads to a natural separation between model formulation and data gathering stored in databases. Hence, the main features of these algebraic modeling languages are:

- Purely declarative representation of the problem,
- Clear separation between formulation and solution,
- Clear separation between model structure and model data.

It is, however, naive to think that one only needs to formulate a problem in a concise declarative form and to link it somehow to a solver in order to solve it. First of all, the "linking process" is not so straightforward as it seems initially. Second, a solver may not exist which could solve the problem at hand in an efficient way. One only needs to look at Fermat's last conjecture which can be stated in a declarative way as $\{a, b, c, n \in N^+ \mid a^n + b^n = c^n, a, b, c \geq 1, n > 2\}$ to convince himself of this fact. Even worse, one can state a problem declaratively for which no solver can exist. This is true already for the rather limited declarative language of first order logic, for which no algorithm exists which decides whether a formula is true or false in general (see [5]).

In this sense, efforts are under way actually in the design of such languages which focus on flexibly linking the declarative formulation to a specific solver to make this paradigm of purely declarative formulation more powerful. This language-solver-interface problem has different aspects and research goes in many directions. A main effort is to integrate symbolic model transformation rules into the declarative language in order to generate formulations which are more useful for a solver. AMPL, for example, automatically detects partially separable structure and computes second derivatives [8]. This information are also handed over to a non-linear solver. LPL, to cite a very different undertaking, has integrated a set of rules to translate *symbolically* logical constraints into 0–1 constraint [11]. To do this in an intelligent way is all but easy, because the resulting 0–1 formulation should be as sharp as possible. This translation is useful for large mathematical models which must be extended by a few logical conditions. For many applications the original model becomes straightforward while the transformed is complicated but still relatively easy to solve (examples were given in [11]). Even if the resulting formulation is not solvable efficiently, the modeler can gain more insights into the structure of the model from such a symbolic translation procedure, and eventually modify the original formulation.

Second generation of modeling languages. Another research activity, actually under way, goes in the direction of extending the algebraic modeling languages in order to express also algorithmic knowledge. This is necessary, because even if one could link an purely declarative language to any solver, it remains doubtful of whether this can be done efficiently in all cases. Furthermore, for many problems it is not useful to formulate them in a declarative way: the algorithmic way is more straightforward and easier to understand. For still other problems a mixture of declarative and algorithmic knowledge leads to a superior formulation in terms of understandability as well as in terms of efficiency, (examples are given below to confirm this findings).

Therefore, AIMMS integrates control structures and procedure definitions. GAMS, AMPL and LPL also allow the modeler to write algorithms powerful enough to solves models repeatedly.

A theoretical effort was undertaken in [10] to specify a modeling language which allows the

modeler (or the programmer) to combine algorithmic and declarative knowledge within the same language framework without intermingle them. The overall syntax structure of a model (or a program) in this framework is as follows:

```
MODEL ModelName
  <declarative part of the model>
BEGIN
  <algorithmic part of the model>
END ModelName.
```

Declarative and algorithmic knowledge are clearly separated. Either part can be empty, meaning that the problem is represented in a purely declarative or in a purely algorithmic form. The declarative part consists of the basic building blocks of declarative knowledge: variables, parameters, constraints, model checking facilities, and sets (that is a way to "multiply" basic building blocks). This part may also contain "ordinary declarations" of an algorithmic language (e.g., type and function declarations). Furthermore, one can declare whole models within this part, leading to nested model structures, which is very useful in decomposing a complex problem into smaller parts. The algorithmic part, on the other hand, consists of all control structures which make the language Turing complete. One may imagine his or her favourite programming language being implemented in this part. A language which combines declarative and algorithmic knowledge in this way is called *modeling language*.

Definition: A *modeling language* is a notational system which allows one to combine (not to merge) declarative and algorithmic knowledge in the same language framework. The content captured by such a notation is called *model*.

Such a language framework is very flexible. Purely declarative models are linked to external solvers to be solved; purely algorithmic models are programs, that is algorithms + data structures, in the ordinary sense.

Modeling language and constraint logic programming. Merging declarative and algorithmic knowledge is not new, although it is not very common in language design. The only existing language paradigm doing it is *constraint logic programming* (CLP), a refinement of logic programming [13]. There are, however, important differences between the CLP paradigm and the paradigm of modeling language as defined above.

(1) In CLP the algorithmic part – normally a search mechanism – is behind the scene and the computation is intrinsically coupled with the declarative language itself. This could be a strength because the programmer does not have to be aware of how the computation is taking place, he or she only writes the rules in a descriptive, that is declarative, way and triggers the computation by a request. In reality, however, it is an important drawback, because – for most nontrivial problem – the programmer *must* be aware on how the computation is taking place. Therefore, to guide the computation in CLP, the declarative program is interspersed with additional rules which have nothing to do with the description of the original problem. In a modeling language, the user either links the declarative part to an external solver or writes the solver within the language. In either case, both parts are strictly separated. Why is this separation so important? Because it allows the modeler to "plug in" different solvers without touching the overall model formulation.

(2) The second difference is that the modeling language paradigm lead automatically to modular design. This is probably to hottest topic in software engineering: building components. Software engineering teaches us that a complex structure can be only managed efficiently by break it down into many relatively independent components. The CLP approach leads more likely to programs that are difficult to survey and hard to debug and to maintain, because such considerations are entirely absent within the CLP paradigm.

(3) On the other hand, the community of CLP has developed methods to solve specific classes of combinatorial problems which seems to be superior to other methods. This is because they rely on propagation, simplification of constraints, and various consistency techniques. In this sense, CLP solvers could be used and linked

with modeling languages. Such a project is actually under way between the AMPL language and the ILOG solver [6],[17].

Hence, while the *representation* of models is probably best done in the language framework of modeling languages, the solution process can taken place in a CLP solver for certain problems.

Modeling examples. Five modeling examples are chosen from very different problem domains to illustrate the highlights of the presented paradigm of modeling language. The first two examples show that certain problems are best formulated using algorithmic knowledge, the next two examples show the power of a declarative formulation, and a last example indicates that mixing both paradigms is sometimes more advantageous.

Sorting. Sorting is a problem which is preferably expressed in an algorithmic way. Declaratively, the problem could be formulated as follows: Find a permutation π such that $A_{\pi_i} \leq A_{\pi_{i+1}}$ forall $i \in \{1 \ldots n-1\}$ where $A_{1\ldots n}$ is an array of objects on which an order is defined. It is difficult to imagine a "solver" that could solve this problem as efficiently as the best known sorting algorithms such as Quicksort, of which the implementation is straightforward.

The reason why the sorting problem is best formulated as an algorithm is probably that the state space is exponential in the number of items, however, the best algorithm only has complexity $O(n \log n)$.

The n-queens problem. The n-queens problem is to place n queens on a chessboard of dimension $n \times n$ in such a way, that they cannot beat each other. This problem can be formulated declarative as follows: $\{x_i, x_j \in \{1 \ldots n\} \mid x_i \neq x_j, x_i + i \neq x_j + j, x_i - i \neq x_j - j\}$, where x_i is the column position of the i-th queen (i.e. the queen in row i).

Using the LPL [12] formulation:

```
MODEL nQueens;
  PARAMETER n; SET i ALIAS j ::= {1..n};
  DISTINCT VARIABLE x{i} [1..n];
  CONSTRAINT S{i,j|i<j} :
     x[i]+i <> x[j]+j AND x[i]-i <> x[j]-j;
END
```

the author was able to solve problems for $n \leq 8$ using a general MIP solver. The problem is automatically translated into a 0–1 problem by LPL. Replacing the MIP-solver by a tabu search heuristic, problems with $n \leq 50$ were solvable within the LPL framework. Using the constraint language OZ [19] problems of $n \leq 200$ are efficiently solvable using techniques of propagation and variable domain reductions. However, the success of all these methods seems to be limited compared to the best we can attain. In [20, 21], Sosic Rok and Gu Jun presented a polynomial time local heuristic that can solve problems of $n \leq 3'000'000$ in less than one minute. The presented algorithm is very simple. The conclusion seems to be for the n-queens problem that an algorithmic formulation is advantageous.

A two person game. Two players choose at random a positive number and note it on a piece of paper. They then compare them. If both numbers are equal, then neither player gets a payoff. If the difference between the two numbers is one, then the player who has chosen the higher number obtains the sum of both; otherwise the player who has chosen the smaller number obtains the sum of both. What is the optimal strategy for a player, i.e. which numbers should be chosen with what frequencies to get the maximal payoff? This problem was presented in [9] and is a typical two-persons zero-sum game. In LPL, it can be formulated as follows:

```
MODEL Game "A finite two-person zero-sum game";
  SET i ALIAS j := /1:50/;
  PARAMETER p{i,j}:= IF(j>i,IF(j=i+1,-i-j,
     MIN(i,j)),IF(j<i,-p[j,i],0));
  VARIABLE   x{i};
  CONSTRAINT  R: SUM{i} x[i] = 1;
  MAXIMIZE  gain: MIN{j} (SUM{i} p[i,j]*x[i]);
END Game.
```

This is an very compact way to declaratively formulate the problem and it is difficult to imagine how this could be achieved using algorithmic knowledge alone. It is also an efficient way to state the problem, because large instances can be solved by an linear programming solver. LPL automatically transforms it into an linear program. (By the way, the problem has an interesting solution: Each player should only choose number smaller than six.)

Equal circles in a square. The problem is to find the maximum diameter of n equal mutually disjoint circles packed inside a unit square.

In LPL, this problem can be compactly formulated as follows:

```
MODEL circles "pack equal circles in a square";
PARAMETER n "number of circles";
SET i ALIAS j = 1..n;
VARIABLE
  t "diameter of the circles";
  x{i}[0,1] "x-position of the centre";
  y{i}[0,1] "y-position  of the centre";
CONSTRAINT
  R{i,j|i<j} "circles must be disjoint":
     (x[i]-x[j])^2+(y[i]-y[j])^2 >= t;
MAXIMIZE obj "maximize diameter": t;
END
```

Maranas al. [15] obtained the best known solutions for all $n \leq 30$ and, for $n = 15$, an even better one using an equivalent formulation in GAMS and linking it to MINOS [16], an well-known non-linear solver.

The (fractional) cutting stock problem. Paper is manufactured in rolls of width B. A set of customers W orders d_w rolls of width b_w (with $w \in W$). Rolls can be cut in many ways, every subset $P' \subseteq W$ such that $\sum_{i \in P'} y_i b_i \leq B$ is a possible cut-pattern, where y_i is a positive integer. The question is how the initial roll of width B should be cut, that is, which patterns should be used, in order to minimize the overall paper waste. A straightforward formulation of this problem is to enumerate all patterns, each giving a variable, then to minimize the number of used patterns while fulfilling the demands. The resulting model is a very large linear program which cannot be solved.

A well-known method in operations research to solve such kind of problems is to use a column generation method (see [3] for details), that is, a small instance with only a few patterns is solved and a rewarding column – a pattern – is added repeatedly to the problem. The new problem is then solved again. This process is repeated, until no pattern can be added. To find a rewarding pattern, another problem – named a Knapsack problem – must be solved.

The problem can be formulated partially be algorithmic partially by declarative knowledge. It consists of two declaratively formulated problems (a linear program and an Knapsack problem), which are both repeatedly solved. In a pseudo-code one could formulate the algorithmic knowledge as follows:

```
SOLVE the small cutting stock problem
SOLVE the Knapsack problem
WHILE a rewarding pattern was found DO
  add the pattern to the cutting stock problem
  SOLVE the cutting stock problem again
  SOLVE the Knapsack problem again
ENDWHILE
```

The two models (the cutting stock problem and the Knapsack problem) can be formulated declaratively. In the proposed framework of modeling language, the complete problem can now be expressed as follows:

```
MODEL CuttingStock;
  MODEL Knapsack(i,w,p,K,x,obj);
    SET i;
    PARAMETER w{i}; p{i}; K;
    INTEGER VARIABLE x{i};
    CONSTRAINT R : SUM{i} w*x <= K;
    MAXIMIZE obj : SUM{i} p*x;
  END Knapsack.
  SET
    w "rolls ordered"; p "possible patterns";
  PARAMETER
    a{w,p} "pattern table";
    d{w}   "demands";
    b{w}   "widths of ordered rolls";
    B      "initial width";
    INTEGER y{w}  "new added pattern";
    C      "contribution of a cut";
  VARIABLE
    X{p}   "number of rolls cut according to p";
  CONSTRAINT
    Dem{w}  : SUM{p} a * X >= d;
  MINIMIZE obj : SUM{p} X;
BEGIN
  SOLVE;
  SOLVE Knapsack(w,b,Dem.dual,B,y,C);
  WHILE (C > 1) DO
    p := p + {'pattern_'+str(#p)};
    a{w,#p} := y[w];
    SOLVE;
    SOLVE Knapsack(w,b,Dem.dual,B,y,C);
  END;
END CuttingStock.
```

This formulation has several remarkable properties. (1) It is short *and* readable. The declarative part consists of the (small) linear cutting stock problem, it also contains, as a submodel,

a Knapsack problem. The algorithmic part implements the column generation method. Both parts are entirely separated. (2) It is a complete formulation, except from the data. No other code is needed; both models can be solved using a standard MIP solver (since the Knapsack problem is small in general). (3) It has a modular structure. The Knapsack problem is an independent component with its own name space; there is no interference with the surrounding model. It could even be declared outside the cutting stock problem. (4) The cutting stock problem is only one problem of a large class of relevant problems which are solved using a column generation or, alternatively, a row-cut generation.

Conclusion. It has been shown that certain problems are best formulated as algorithms, others in a declarative way, still others need both paradigms to be stated concisely. Computer science made available many algorithmic languages; they can be contrasted to the algebraic modeling languages which are purely declarative. A language, called modeling language, which combines both paradigms was defined in this paper and examples were given showing clear advantages of doing so. Its is more powerful than both paradigms separated.

However, the integration of algorithmic and declarative knowledge cannot be done in an arbitrary way. The language design must follow certain criteria well-known in computer science. The main criteria are: *reliability* and *transparency*. Reliability can be achieved by a unique notation to code models, that is, by a modeling language, and by various checking mechanisms (type checking, unit checking, data integrity checking and others). Transparency can be obtained by flexible decomposition techniques, like modular structure as well as access and protection mechanisms of these structure, well-known techniques in language design and software engeneering.

Solving efficiently and relevant optimization problems using present desktop machine not only asks for fast machines and sophisticated solvers, but also for formulation techniques that allow the modeler to communicate the model easily and to build it in a readable and maintainable way.

References

[1] BISSCHOP, J.: *AIMMS, The Modeling System*, Paragon Decision Technology B.V.,(www.paragon.nl), 1998.

[2] BROOKE, A., KENDRICK, D., AND MEERAUS, A.: *GAMS, A User's Guide*, The Scientific Press, 1988.

[3] CHVÁTAL, V.: *Linear Programming*, W.H. Freeman Company, New York, 1973.

[4] FEIGENBAUM, E.A.: 'How the 'What' Becomes the 'How", *Communications of the ACM* **39:5** (1996), 97–104.

[5] FLOYD, R.W., AND BEIGEL, R.: *The Language of Machines, An Introduction to Computability and Formal Languages*, Computer Science Press, 1994.

[6] FOURER, R.: 'Extending a general-purpose algebraic modeling language to combinatorial optimization: a logic programming approach': *D.L. Woodruff (ed.), Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search: Interfaces in Computer Science and Operations Research*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998, pp. 31–74.

[7] FOURER, R., GAY, D.M., AND KERNIGHAN, B.W.: *AMPL, A Modeling Language For Mathematical Programming*, The Scientific Press, San Francisco, 1993.

[8] GAY, D.M.: 'Automatically Finding and Exploiting Partially Separable Structure in Nonlinear Programming Problems', *AT&T Bell Laboratories, Murray Hill, New Jersey* (1996).

[9] HOFSTADTER, D.R.: *Metamagicum, Fragen nach der Essenz von Geist und Struktur*, Klett-Cotta, Stuttgart, 1988.

[10] HÜRLIMANN, T.: *Computer–Based Mathematical Modeling, Habilitation Script*, Faculty of Economic and Social Sciences, Institute of Informatics, University of Fribourg, December, 1997.

[11] HÜRLIMANN, T.: *An Efficient Logic-to-IP Translation Procedure, Working Paper, March*, Institute of Informatics, University of Fribourg, (ftp://ftp-iiuf.unifr.ch/pub/lpl/doc/APMOD1.ps), 1998.

[12] HÜRLIMANN, T.: *Reference Manual for the LPL Modeling Language, Working Paper, Version 4.30, June*, Institute of Informatics, University of Fribourg, (ftp://ftp-iiuf.unifr.ch/pub/lpl/doc/Manual.ps), 1998.

[13] JAFFAR, J., AND MAHER, M.J.: *Constraint Logic Programming: A Survey*, (see at ftp://pop.cs.cmu.edu/usr/joxan/public), 1996.

[14] LOUDEN, K.C.: *Programming Languages – Principles and Practice*, PWS–KENT Publ. Comp., 1993.

[15] MARANAS, C.D., FLOUDAS, C.A., AND PARDALOS, P.M.: *New results in the packing of equal circles in a square*, Departemnt of Chemical Engineering, Princeton University, 1993.

[16] MURTAGH, B.A., AND SAUNDERS, M.A.: *MINOS 5.0, User Guide*, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1987.

[17] SA, ILOG: *ILOG Solver 4.0 User's Manual; ILOG Solver 4.0 Reference Manual*, ILOG, Inc., Mountain View CA, 1997.

[18] SCHRAGE, L.: *Optimization modeling with LINGO*, Lindo Systems, Inc., (www.lindo.com), 1998.

[19] SMOLKA, G.: 'The Oz Programming Model', *van Leeuwen J. (ed.), Computer Science Today, Lecture Notes in Computer Science* **1000** (1995), 324–343.

[20] SOSIC, R., AND GU, J.: 'A polynomial time algorithm for the n-queens problem', *SIGART Bulletin* **1:3** (1990), 7–11.

[21] SOSIC, R., AND GU, J.: '3,000,000 queens in less than one minute', *SIGART Bulletin* **2:1** (1991), 22–24.

Tony Hürlimann
Institute of Informatics
University of Fribourg, Switzerland
E-mail address: tony.huerlimann@unifr.ch

AMS 1991 Subject Classification: D.3.2.
Key words and phrases: algorithmic language, declarative language, modeling language, solver.