Algebraic Modeling Languages for Optimization

Robert Fourer
Northwestern University

Algebraic modeling languages are sophisticated software packages that provide a key link between an analyst's mathematical conception of an optimization model and the complex algorithmic routines that seek out optimal solutions. By allowing models to be described in the high-level, symbolic way that people think of them, while automating the translation to and from the quite different low-level forms required by algorithms, algebraic modeling languages greatly reduce the effort and increase the reliability of formulation and analysis. They have thus played an essential role in the spread of optimization to all aspects to OR/MS and to many allied disciplines.

Background and motivation

Practical software packages for solving optimization problems emerged in the 1950s, as soon as there were computers to run them. Initially based on linear programming, these "solvers" were soon generalized to allow for nonlinearities and to accommodate integer variables and other discrete decisions. Despite continuing progress in algorithms and in computing, however, by the beginning of its second decade large-scale optimization had come to be seen as failing to live up to its potential. The key weakness in early optimization systems was not in their algorithms, however, but in their interaction with modelers. The human time and cost of preparing a solver's input and examining its output often greatly dominated the computer costs of solving. The cause of this difficulty, and its ultimate cure, can best be understood by considering the steps of the optimization modeling process and their interaction with the technical requirements of large-scale optimization.

The process of building practical optimization models involves several interrelated steps. The first and most important is extensive communication with the owner of a decision problem to identify the problem ingredients and to ascertain the extent to which optimization is feasible within the managerial structure of the client organization and the cognitive limitations of the model user. Next is the formulation of a mathematical abstraction of the problem — a *model* — that offers a sufficiently accurate characterization of the real situation in terms of reasonably available data. Further steps build datasets, generate the corresponding optimization problem instances, feed the problem instances to solvers, run the solvers to produce results that are optimal or near-optimal by the model's criteria, and process the results into descriptions of decisions in forms that clients can understand and analyze. These tasks are carried out repeatedly in a kind of

feedback loop, as further communication results in model modifications and data refinements due to invalid assumptions, bad data, programming errors, and (most interestingly) the identification of previously unelucidated policies, constraints and preferences. The success of an optimization application depends critically on how fast one can implement the central feedback loop — formulation, solution, analysis, revision. The faster these steps, the greater the likelihood that the modeling effort will receive sufficient attention from the client in the communication phase to ensure that the model will eventually be adopted and supported. Thus as the number-crunching solution phase became progressively more efficient with advances in algorithms and computers, the steps involving human analysts became the bottlenecks in this process.

In fact the optimization development cycle was found to take much more analyst time than expected. The culprit was the awkward and error-prone work of converting an optimization problem between the modeler's conception and the algorithm's representation. Indeed the natural way for a modeler to think about and express models is in direct conflict with the input requirements of solution algorithms. As detailed in Fourer (1983), whereas the modeler's form is symbolic, general, concise, and understandable to other modelers, the solver's form is contrary in every respect: explicit, specific, extensive, and convenient for computation. For all but the smallest and simplest instances, the only practical way to make the conversion from the modeler's to the algorithm's form is by writing a computer program for the purpose, and it was the continued maintenance and debugging of this program in successive cycles of the development process that unexpectedly soaked up so much analyst time. Whether a program of this kind is working correctly is particularly hard to confirm, as the only detailed evidence of its performance consists of voluminous coefficient lists and other details that are specifically intended for algorithmic efficiency rather than human comprehension.

Optimization modeling languages were conceived as a way of alleviating this bottleneck of conversion. They allow people to convey their formulations to computer systems in much the same way that they would write them out or describe them to colleagues. Computer systems that implement modeling languages also facilitate analysis and reporting using the terminology of the model, thus further speeding the development cycle.

Any convenient form of representation for some class of optimization applications can in principle give rise to a modeling language. However many general-purpose modeling languages are based on the familiar mathematical representation of an optimization problem as the minimization or maximization of a function of decision variables, subject to equations and inequalities in functions of the variables. The most popular languages are founded in particular on familiar

expressions — like $\sum_{j=1}^{n} a_{ij}\, x_j$, $\sqrt{\sum_{s \in S} (g_{rs} - h_s)^2}$, or $G_{km} \cos(\delta_k - \delta_m)$ — that use the operators and functions of elementary algebra, though written in a form that requires only a computer character set. Most such languages have been generalized through the use of notations from logic, computer programming, and other disciplines, but in recognition of their origins they are widely known as *algebraic* modeling languages (Bisschop and Meeraus 1982).

The initial popularity of algebraic modeling languages derived in part from their users' familiarity with mathematical optimization theory. However they quickly became recognized as offering a valuable tradeoff between the convenience of highly application-specific representations and the power of informal natural-language problem descriptions. Their combination of precision and generality enabled them to support optimization as a paradigm for modeling and decision-making in diverse applications of operations research and throughout engineering, science, economics, and management. At the same time their flexibility enabled them to accommodate the unique features that distinguish individual applications in realistic situations.

Example

To give a further view of the issues involved in designing, selecting, and using an algebraic modeling language, we present a modest example of a model of optimal multiperiod transportation of a single commodity. Our presentation describes the model first in words and mathematical formulas, and then equivalently in one of the widely used modeling languages. We conclude by describing three major aspects of working with the model: the specification of data, the invocation of solvers, and the examination of results.

Mathematical formulation. To begin describing an algebraic model for transportation, we may say that we have a set *I* of cities where supply of a product originates, and a set *J* of cities where demand must be met. A set of "links" $L \subseteq I \times J$ specifies those origin-destination pairs (i, j) for which shipments from *i* to *j* are possible. We want to plan for the next *T* weekly time periods.

The objective of this model is to decide how much to ship from each origin to each destination in each week, so as to minimize the total cost of all shipments. Thus we introduce decision variables $x_{ijt} \geq 0$ and parameters $c_{ijt} > 0$ for each $(i, j) \in L$ and $t = 1, \ldots, T$, representing respectively the amounts to be shipped (which will be determined by optimization) and the costs per unit of shipment (which are supplied as data). In terms of these quantities, the objective may be written algebraically as

$$\text{Minimize } \sum_{(i,j) \in L} \sum_{t=1}^{T} c_{ijt}\, x_{ijt}$$

The essential constraints on the decision variables are next described in terms of parameters a_{it} for each $i \in I$ and $t = 1, \ldots, T$, representing the amount that becomes available for shipment at origin *i* in week *t,* and b_{jt} for each $j \in J$ and $t = 1, \ldots, T$, representing the amount required to meet expected demands at destination *j* in week *t.* The possibility of week-to-week fluctuations in shipping costs suggests that not all supply should be shipped out in the week that it becomes available. Thus we also introduce decision variables y_{it} for each $i \in I$ and $t = 1, \ldots, T$, to represent the amount of product in inventory at origin *i* at the end of week *t.* The following algebraic constraints then serve to express the limitations on shipping out of each origin and the requirements of meeting demand at each destination:

$$\sum_{j \in J:(i,j) \in L} x_{ijt} + y_{it} \leq a_{it} + y_{i,t-1}, \text{ for each } i \in I, t = 1, \ldots, T$$

$$\sum_{i \in I:(i,j) \in L} x_{ijt} = b_{jt}, \text{ for each } j \in J, t = 1, \ldots, T$$

For the sake of this simple model we disregard the possibility of initial inventories, thus implicitly setting to zero all terms y_{i0} in the origin constraints for $t = 1$.

The shipment plan is also commonly subject to certain operational considerations. As just one example, the amount shipped over link (i, j) in all weeks may be required to sum to at least a certain amount, given by a parameter d_{ij}, if that link is used in any period at all. A quite general way of implementing this restriction through algebraic constraints is by defining a corresponding collection of decision variables z_{ij} that can only take the values 0 or 1. Then we may write

$$d_{ij} z_{ij} \leq \sum_{t=1}^{T} x_{ijt} \leq \min\left(\sum_{t=1}^{T} a_{it}, \sum_{t=1}^{T} b_{jt}\right) z_{ij}, \text{ for each } (i,j) \in L$$

which forces shipments to be zero when z_{ij} is zero, or to be at least d_{ij} (and at most some implied upper bound) when z_{ij} is one.

Modeling language formulation. The representation of this model in an algebraic modeling language is fundamentally the same as this mathematical formulation, with the differences deriving mainly from the need to communicate the model unambiguously and to use a standard character set. Thus for instance in the AMPL modeling language (Fourer, Gay and Kernighan 1990) the sets and parameters that describe the data might be specified as follows:

```
set ORIG;   # origins
set DEST;   # destinations

set LINKS within {ORIG,DEST};

param T integer > 0;

param supply {ORIG,1..T} >= 0;
param demand {DEST,1..T} >= 0;

param cost {LINKS,1..T} > 0;
param minShip {LINKS} >= 0;
```

AMPL defines a standard indexing expression such as `{ORIG,1..T}` to correspond to a statement like "for each $i \in I$, $t = 1, \ldots, T$" in the mathematical formulation (though the *i* and *t* need be included only where actually used). The use of more meaningful names like `ORIG` for *I* and `supply` for *a,* while not required, often proves useful for keeping models understandable as they grow in complexity. Models can also be more thoroughly documented through a variety of comments, which are seen here for the first two sets but will be otherwise omitted in this description for the sake of brevity.

Decision variables are next defined in much the same way as parameters:

```
var Ship {LINKS,1..T} >= 0;
var Inv {ORIG,1..T} >= 0;

var Use {LINKS} integer >= 0, <= 1;
```

Indeed the only difference between parameters and variables is that the former are specified as part of the data while the latter are given their values by the solver so as to optimize the objective. Given the definitions in this example, AMPL's statement for the objective of the model is as follows:

```
minimize TotalCost:
   sum {(i,j) in LINKS, t in 1..T} cost[i,j,t] * Ship[i,j,t];
```

This is the same algebraic expression as in the mathematical formulation, adapted for input to a computer system; `sum {...}` corresponds to the sigma expressions, while `cost[i,j,t]` and `Ship[i,j,t]` are the AMPL representations for c_{ijt} and x_{ijt}. An explicit operator `*` is introduced to represent the multiplication that is customarily implicit in mathematical expressions.

Constraints are similarly converted to algebraic expressions in the modeling language. They are somewhat more complex than the objective because they come in indexed collections and use relational operators for equalities and inequalities:

```
subject to Supply {i in ORIG, t in 1..T}:
   sum {(i,j) in LINKS} Ship[i,j,t] + Inv[i,t]
      <= supply[i,t] + (if t > 1 then Inv[i,t-1] else 0);

subject to Demand {j in DEST, t in 1..T}:
   sum {(i,j) in LINKS} Ship[i,j,t] = demand[j,t];

subject to ZeroMin1 {(i,j) in LINKS}:
   minShip[i,j] * Use[i,j] <= sum {t in 1..T} Ship[i,j,t];

subject to ZeroMin2 {(i,j) in LINKS}:
   sum {t in 1..T} Ship[i,j,t] <=
      min (sum {t in 1..T} supply[i,t],
           sum {t in 1..T} demand[j,t]) * Use[i,j];
```

The emphasis is on keeping the original forms of the constraints as much as possible, while letting the AMPL translator automate the work of evaluating

coefficient expressions, collecting terms on the left, and other regularizations that may be required by solvers. Each modeling language does introduce some changes; here AMPL requires the double-inequality constraint to be split in two, but streamlines the specifications of the supply and demand constraints by interpreting `{(i,j) in LINKS}` so that it specifies indexing over only whichever index has not already been fixed. Also the assumption of zero initial inventories must be made explicit, in this example by using an if-then-else construct to handle inventories at the end of "week 0" specially.

Specification of data. Each modeling language offers its own format for associating actual data values with the sets and parameters in the symbolic model. A small collection of data for our example could be specified by an AMPL text file that begins as follows:

```
set ORIG := YYZ LAF CVG PIT CLE ;
set DEST := ABE ORF ;

set LINKS := (YYZ,ABE) (YYZ,ORF) (LAF,ABE) (CVG,ORF)
             (PIT,ABE) (PIT,ORF) (CLE,ABE) (CLE,ORF) ;

param T := 5 ;

param demand:     1     2     3     4     5 :=
          ABE  1000  1200  1900  2500  2000
          ORF  2100  3000  4900  7700  5000 ;

param supply:     1     2     3     4     5 :=
          YYZ  2100  2250  3190  3120  3500
          LAF  1400  1250  1320  1220  1100
          CVG  1650  1250  2290  2120  2300
          .......
```

Model and data together specify a particular *instance* of an optimization problem for which a solution can be sought.

Modeling language systems typically also offer facilities for exchange of data with popular databases, spreadsheets, and other repositories of data for decision support. Indeed there is a close correspondence between the way that data values are described in algebraic models and the way they are organized in relational databases (Fourer 1997). Close interaction with data management software is often important to the integration of optimization into business operations.

Invocation of solvers. Modeling language software automatically reads and interprets a model and data, generates an instance, and conveys the instance to a solver in the required form. In AMPL it suffices to give only a few commands for these purposes:

```
ampl: model multiEORMS.mod;
ampl: data multiEORMS.dat;
ampl: option solver cplexamp;
ampl: solve;

73 variables:
       8 binary variables
       65 linear variables
51 constraints, all linear; 221 nonzeros
1 linear objective; 40 nonzeros.

CPLEX 12.2.0.2: optimal integer solution; objective 288503.5
65 MIP simplex iterations
2 branch-and-bound nodes
```

The solver software is a separate product for which there may be many alternatives. For this model, a different mixed-integer programming solver might have been used instead:

```
ampl: model multiEORMS.mod;
ampl: data multiEORMS.dat;
ampl: option solver gurobi;
ampl: solve;

Gurobi 4.0.1: optimal solution; objective 288503.5
71 simplex iterations
plus 52 simplex iterations for intbasis
```

Also a full variety of options, specific to each solver, are accessible as needed to set algorithmic options and report progress of long runs.

Examination of results. Once the solver has returned a solution, the same expression forms that are so convenient in describing the model to the computer system can also be used to describe the results to be viewed. For example to show for each link the ratio of total shipments to minimum shipment over all periods, one can simply ask AMPL to "display" the appropriate sum, adapting the same summation syntax that was used in the model:

```
ampl: display {(i,j) in LINKS}
ampl?    sum {t in 1..T} Ship[i,j,t] / minShip[i,j];

:       ABE       ORF   :=
CLE   0         1.69032
CVG    .        1.48992
LAF   1.38636    .
PIT   0         0
YYZ   1         2.99143
```

Simple displays of this kind do much to support the cycle of development, by speeding the modeler's aggregation, transformation, and analysis of solutions. For later deployment of the model, facilities are also available for writing more precisely formatted text and for sending results off to other software for analysis.

Advantages

The fundamental concept of algebraic modeling languages — that the entire optimization modeling cycle is best carried out at the level of the model formulation — makes possible the creation of modeling systems that have a number of desirable characteristics. This article has already described how such systems promote optimization modeling by making the entire process more efficient and reliable. The modeling language concept has proven to have other benefits as well. Principally these relate to independent treatment of distinct aspects of optimization, and to extensions well beyond linear optimization.

Independence. In contrast to the highly integrated design common of software for mathematical and statistical modeling and for simulation, modeling language systems for optimization have promoted an independence of model, data, and solvers. This property has proved to be of benefit of users in several ways.

Because the sets and parameters of a model are described symbolically along with the variables, objectives, and constraints, the same model readily describes any number of problem instances of different sizes and purposes. This model-data independence allows prototypes to be scaled up quickly to larger and more realistic scenarios through changes to the data files alone. Equally it provides flexibility to experiment with different formulations on the same data, as is often essential for finding tractable approaches to difficult mixed-integer modeling applications. Following the initial development stages, model-data independence is also beneficial, allowing the model to be frozen while deployment focuses on periodically generating data for new runs. The full symbolic model description remains accessible, however, whenever modifications are necessary to accommodate new circumstances or analyses.

Because modeling languages are designed to describe models and their data in an abstract way, they are not tied to particular software for optimization or even to particular methods. This model-solver independence allows instances to be benchmarked over a range of solvers. The choice of a solver for deployment can then be based on a tradeoff between price and performance, and can be revisited as optimization technology evolves. The very substantial changes in linear optimization packages that have occurred over recent decades have thus not required corresponding changes in modeling language design.

Another virtue of model-solver independence is to relieve the analyst of much tedious work of converting between the modeler's form and the various algorithms' forms. Originally this work consisted mainly of generating coefficient lists and bound vectors. But as languages have become more sophisticated it has come to include conversions to linear representations from other forms that may be closer

to the original model conception, such as piecewise-linear formulations and network node-arc descriptions.

Extensions. Algebraic languages can express nonlinear optimization problems as easily as linear ones, simply by permitting expressions that are nonlinear in the variables. Thus for instance in our transportation example if it is desired to encourage shipments of moderate size, neither too small nor too large, the objective could be changed to

$$\text{Minimize } \sum_{(i,j)\in L} \sum_{t=1}^{T} c_{ijt} \frac{x_{ijt}^{\alpha}}{1-x_{ijt}/\ell_{ij}}$$

where $0 < \alpha < 1$ and ℓ_{ij} is some positive link capacity. To specify this in a modeling language it suffices to write the corresponding nonlinear expression:

```
minimize TotalCost:
   sum {(i,j) in LINKS, t in 1..T}
      cost[i,j,t] * Ship[i,j,t]^alpha / (1 - Ship[i,j,t]/lim[i,j]);
```

After the model and data have been processed, a representation of each nonlinear objective and constraint expression is included as part of the instance representation passed to the solver interface. Then the interface uses this representation to compute function values at successive points generated by the solver as it iterates toward the optimum; the interface also provides analytical (not approximate) first and second derivatives automatically by methods that avoid the overhead of symbolically differentiating each nonlinear expression (Rall and Corliss 1996; Griewank and Walther 2008). This approach is considerably more efficient and less error-prone than working directly with the nonlinear solver, which would require writing, debugging, and maintaining a program for each nonlinear expression and its derivatives.

The technology for recognizing and processing conventional nonlinear expressions extends moreover to virtually any kind of expression that can be written in terms of functions and operators. Thus it is possible to substantially extend the range of models that can be expressed naturally through algebraic modeling languages. Current implementations allow for example the specification of complementarity conditions, and the description of logical restrictions using operators like “or” and “not” rather than through the introduction of zero-one variables. Also special cases like quadratic objectives and constraints can be detected and transformed automatically.

The algebraic expressions that are useful in describing individual objectives and constraints are also useful in describing manipulations of models and transformations of data. Thus almost as soon as modeling languages became available, users started finding ways to adapt model notations to implement sophisticated solution strategies and iterative schemes. These efforts stimulated the

evolution within algebraic modeling languages of scripting features, which include statements for looping, testing, and assignment. Thus for instance to test the sensitivity of our multiperiod transportation model to the minimal-shipment thresholds, the modeler could write a simple loop:

```
for {k in 1..10} {
   let {(i,j) in LINKS} minShip[i,j] := minShip[i,j] + 250;
   solve;
   if solve_result = "infeasible" then break;
}
```

Industrial and research applications now commonly employ scripts involving many hundreds of lines. The efficiency and convenience of algebraic modeling is thus extended to situations much more complex than the solving of individual optimization problems.

Alternatives

The ideas and benefits of algebraic modeling languages are available to various extents in several kinds of software.

General-purpose algebraic modeling languages embody model-data-solver independence to the greatest degree, supporting links to numerous independently-developed solvers and data-management systems. The most widely used commercial systems in this category are AIMMS (Paragon Decision Technology 2011), AMPL (AMPL Optimization 2011), GAMS (GAMS Development 2011), and MPL (Maximal Software 2011); for noncommercial uses, GNU MathProg (Free Software Foundation 2011) and Zimpl (Zuse Institute 2011) are open-source alternatives licensed under the GNU GPL. All base their language designs on the same fundamental ideas, though with varying specifics in some key respects. They differ more substantially in aspects of their user, solver, and data management interfaces.

Solver-specific algebraic modeling languages offer similar designs but have been implemented to be used mainly or exclusively with one solver developer's products. By forgoing solver independence, they can offer more complete and integrated support for one suite of solvers, often including ones that go beyond the traditional algorithmic approaches for linear and smooth nonlinear problems. Among the best-known are LINGO (LINDO Systems 2011a), Mosel (Fair Isaac 2011), and OPL (IBM Corporation 2011b).

An algebraic modeling framework for optimization can also be implemented within a general object-oriented modeling language. Specialized object types are defined to represent common model entities such as parameters, variables, and constraints; then all of the standard operators and functions are overloaded to act

specially when applied to these types. Thus for example using the CPLEX Optimization Studio's Concert C++ library (IBM Corporation 2011a) one can make definitions such as

```
IloEnv env;
IloNumArray supply(env);
IloNumVarArray Use (env, nOrig, 0, 1, ILOINT);
IloExpr totalShipFrom(env);
```

and then express, for example, some supply-limit constraints by writing

```
for (i = 0; i < nOrig; i++) {
   for (j = 0; j < nDest; j++) {
      totalShipFrom += Ship[i][j];
   }
   mod.add(totalShipFrom <= supply[i] * Use[i]);
}
```

What appear to be arithmetic and comparison operations are in fact interpreted as instructions to build up a constraint data structure for an affiliated solver. A similar Concert interface is available for Java and .NET, and the same idea with more general-purpose solver support has been carried through by, among others, FLOPC++ (COIN-OR 2011), OptimJ (Ateji 2011) for Java, and Pyomo (Sandia 2011) for Python. Compared to languages specially designed for algebraic modeling, these object-oriented tools have less natural representations — particularly in the use of indexing sets — and require more user involvement in the lower-level aspects of programming. However they can offer the advantages of a much richer programming environment than is afforded by the scripting facilities of specialized modeling languages; also they hold out the possibility of simplifying the integration of optimization models into broader applications.

Several kinds of modeling language integration with general-purpose analytical tools have also proved popular. Some general-purpose modeling languages have connections to solvers running under MATLAB (Mathworks 2011), and there is a MATLAB-based connection from AMPL to many independent solvers through the TOMLAB environment (Tomlab Optimization 2011). The AMPL-like OPTMODEL language (SAS Institute 2011) supports SAS/OR solvers as an integrated part of the SAS business analytics system. By far the most popular are modeling languages implemented as Microsoft Excel add-ins, notably the Frontline Premium Solver (Frontline Systems 2011) and What'sBest (LINDO Systems 2011b), with a variety of solver options. Because these languages are closely tied to Excel spreadsheet data forms, they are very limited in power and expressiveness. However they offer the very substantial benefit of being able to integrate optimization into the most widely used environment for all kinds of business analyses.

Extensions

Enhancements to algebraic modeling languages are basically of two kinds: extensions to the languages themselves, and improvements to the ways in which the languages interact with other systems.

Modeling language extensions tend to be driven by solver developments. Whenever algorithms are developed to effectively solve new forms of optimization problems, modeling language developers are challenged to provide more convenient support. Operators and syntaxes may be added to let modelers describe new forms in the most natural ways, as happened for example with the advent of more effective solution strategies to handle complementarity conditions (Ferris, Fourer and Gay 1999). Alternatively, additional logic may be introduced to detect special cases that are significant to new algorithms, as occurred with the discovery of efficient methods for second-order cone problems that were equivalent to several common kinds of nonlinear constraints (Lobo et al. 1998); here the recognition technology is still in the process of being developed. Ferris et al. (2009) survey a variety of such problems including also bi-level and generalized nonlinear optimization. Constraint programming solvers have motivated a variety of extended forms for logical and discrete optimization (Lustig and Puget 2001) which have also proved to be valuable for describing discrete optimization more naturally to other kinds of solvers.

Enhancements to the interfaces and interoperability of modeling language systems tend to be driven by more general developments in computing. This has been seen in the creation of more sophisticated user interfaces for model building, more powerful object-oriented programming interfaces for embedding models within larger applications, and closer links to data management systems. Recent popularity of Python-based optimization modeling tools is an example of such a trend. Another is the growing attractiveness of optimization "software as a service" — as pioneered by the NEOS Server (Czyzyk, Mesnier, and Moré 1998; Dolan et al. 2002) — which seems likely to motivate widespread access to algebraic modeling languages "in the cloud" in ways that will foster even more efficient and convenient development cycles for optimization.

References

AMPL Optimization LLC (2011). AMPL modeling language, `www.ampl.com`.

Ateji SAS (2011). OptimJ modeling language, `www.ateji.com/optimj`.

Bisschop, Johannes and Meeraus, Alexander (1982). "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment." *Mathematical Programming Study* **20,** 1-29.

COIN-OR Foundation (2011). FLOPC++ modeling language, projects.coin-or.org/FlopC++.

Czyzyk, Joseph, Mesnier, Michael P., and Moré, Jorge J. (1998). "The NEOS Server." *IEEE Computational Science and Engineering* **5,** 68-75.

Dolan, Elizabeth D., Fourer, Robert, Moré, Jorge J., and Munson, Todd S. (2002). "Optimization on the NEOS Server." *SIAM News* **35:6,** 4, 8-9.

Fair Isaac Corporation (2011). Xpress-Mosel modeling language, www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Mosel.aspx.

Ferris, Michael C., Dirkse, Steven P., Jagla, Jan-H., and Meeraus, Alexander (2009). "An Extended Mathematical Programming Framework." *Computers and Chemical Engineering* **33,** 1973-1982.

Ferris, Michael C., Fourer, Robert, and Gay, David M. (1999). "Expressing Complementarity Problems in an Algebraic Modeling Language and Communicating Them to Solvers." *SIAM Journal on Optimization* **9,** 991-1009.

Fourer, Robert (1983). "Modeling Languages versus Matrix Generators for Linear Programming." *ACM Transactions on Mathematical Software* **9,** 143-183.

Fourer, Robert (1997). "Database Structures for Mathematical Programming Models." *Decision Support Systems* **20,** 317-344.

Fourer, Robert, Gay, David M., and Kernighan, Brian W. (1990). "A Modeling Language for Mathematical Programming." *Management Science* **36,** 519-554.

Fourer, Robert, Gay, David M., and Kernighan, Brian W. (2003). *AMPL: A Modeling Language for Mathematical Programming,* 2nd edition. Cengage Learning, Belmont, California.

Free Software Foundation (2011). GNU MathProg modeling language, www.gnu.org/software/glpk.

Frontline Systems, Inc. (2011). Premium Solver for Excel, www.solver.com/xlspremsolv.htm.

GAMS Development Corporation (2011). GAMS modeling language, www.gams.com.

Griewank, Andreas, and Walther, Andrea (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation,* 2nd edition. SIAM, Philadelphia, Pennsylvania.

IBM Corporation (2011a). Concert Technology, www.ibm.com/software/integration/optimization/cplex-optimization-studio/modeling/#technology.

IBM Corporation (2011b). OPL modeling language, www.ibm.com/software/integration/optimization/cplex-optimization-studio/modeling/#language.

Josef Kallrath, ed. (2004). *Modeling Languages in Mathematical Optimization.* Kluwer Academic Publishers, Dordrecht, The Netherlands.

Kuip, C.A.C. (1993). "Algebraic Languages for Mathematical Programming." *European Journal of Operational Research* **67,** 25-51.

LINDO Systems (2011a). LINGO modeling language, `www.lindo.com`.

LINDO Systems (2011b). What'sBest Excel add-in, `www.lindo.com`.

Lobo, Miguel Soma, Vandenberghe, Lieven, Boyd, Stephen, and Lebret, Hervé (1998). "Applications of Second-Order cone Programming." *Linear Algebra and its Applications* **284,** 193-228.

Lustig, Irvin J. and Puget, Jean-François (2001). "Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming." *Interfaces* **31:6,** 29-53.

Maximal Software Inc. (2011). MPL modeling language, `www.maximalsoftware.com`.

Paragon Decision Technology (2011). AIMMS modeling language, `www.aimms.com`.

Rall, Louis B. and Corliss, George F. (1996). "An Introduction to Automatic Differentiation." Martin Berz et al., editors, *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, Pennsylvania, pp. 1-17.

Sandia National Laboratories (2011). Pyomo modeling language, `software.sandia.gov/trac/coopr/wiki/Pyomo`.

SAS Institute Inc. (2011). SAS/OR PROC OPTMODEL modeling language, `www.sas.com/technologies/analytics/optimization/or`.

The Mathworks, Inc. (2011). MATLAB technical computing environment, `www.mathworks.com`.

Tomlab Optimization (2011). TOMLAB optimization environment, `www.tomopt.com/tomlab`.

Zuse Institute Berlin (2011). Zimpl modeling language, `zimpl.zib.de`.