Proceedings of the 28th Annual Hawaii International Conference on System Sciences — 1995

Generating Optimization-Based Decision Support Systems

Arthur Geoffrion

Anderson Graduate School of Management
U.C.L.A.
Los Angeles, California 90024

Sergio Maturana

Departamento de Ingeniería de Sistemas
Pontificia Universidad Católica de Chile
Santiago, Chile

Abstract

This paper discusses the implementation of optimization based DSSs. An approach is proposed that will enable OR/MS analysts to develop this kind of system much more efficiently than is possible today. The aim is to develop systems with modern GUIs, that interact with DBMSs, and that can be built with little programming effort. Many of the tools required for this approach were developed to support SML, which was chosen to represent the models because it has ample expressive power, completely specified syntax and semantics, total structure/data independence, and lends itself to the surface/deep structure distinction, which makes the approach feasible. The approach is being implemented in a research project funded by the Chilean Government and several firms and is being tested on problems taken from these firms.

1 Introduction

As many authors have noted (e.g., Schultz and Pulleyblank [1]), better delivery systems are required in order for optimization technology to find its way into practice at a significantly faster rate that at present. One popular approach is to incorporate optimization into systems designed for end users rather than for OR/MS specialists. The trend toward embedding optimizing solvers into spreadsheets provides an example.

This approach is consistent with Little's appealing idea [2] of empowering the organizational front line by giving people systems with which they can solve their own problems, as distinct from the traditional approach of having specialists solve problems and then deliver solutions to the problem owners.

On the other hand, there are many good reasons for having an OR/MS analyst assist the user in the process of modeling. To mention only two: users usually don't have the time to think in depth about their problems, since they are too busy handling them. Second, many people think modeling is an activity that requires a great deal of experience [3]. Most users lack this experience. Gass [4], for example, says "Beware the user as a modeler" noting that the user-modeler will tend to believe in the model and may use the solution without doing the necessary verification and validation.

The OR/MS analyst also plays an important role in the implementation of what we call an *optimization-based decision support system* (e.g., PDSS [5]), which integrates the optimizer with machinery for preparing the necessary data, and also provides a good user interface.

The concern of this paper is with tools that will enable OR/MS analysts and modelers to develop optimization-based decision support systems much more efficiently (i.e., with far less skilled labor) than is possible today. We aim for systems with a modern graphic user interface, with models and outputs that can be modified easily (up to a point) so long as the basic nature of the application stays the same, and that can be built without significant programming effort.

Since attempting to do this in general is probably too ambitious, we restrict ourselves to certain types of problems, namely, production planning, distribution, and inventory. This will be done in the context of the 3 year project funded by the Chilean Government and several Chilean firms, which is described in [6]. The objective of this project is to develop tools for generating this kind of system quickly and inexpensively. Several systems are being developed for the participating firms to test the proposed approach.

2 Requirements for a Good Delivery System

An optimization-based Decision Support System can also be viewed as an application-specific delivery system for an optimizer engine, which is the problem-generation and results analysis/reporting software needed to engage an optimizer engine in useful work. A good application-specific delivery system for a given engine is one that meets these major objectives:

1. It delivers, for the target application, broad functionality relating to problem-generation and results analysis/reporting.
2. It makes efficient enough use of computer resources that it does not compromise the performance of the engine.
3. It is productively usable by people other than those who developed the engine, especially by people who are knowledgeable concerning the target application but not necessarily concerning modeling technology. Among other things, this requires extensive error-control features.
4. It is readily maintainable and evolvable.

Good delivery systems are essential if significant benefits are to be obtained from an optimizer. In their absence, even the finest optimizer would languish in the developer's laboratory because there would be no practical way to engage it in useful work outside of that laboratory.

Unfortunately, good delivery systems for optimizers are very time consuming to build, and their construction makes severe demands on several quite different kinds of skills. Much could be said about why this is so, but anyone with practical experience in this area would agree with this assertion and also with the need for improved methods and tools for building good delivery systems.

The four objectives just listed imply certain requirements that are not difficult to discern.

2.1 The Deep Structure versus Surface Structure Distinction

The first requirement is that a good delivery system needs to make a distinction between a model's *deep structure* and its *surface structure*. A model's *deep structure* is whatever aspects of its mathematical structure truly impact a particular optimizer's ability to solve it. Everything else is *surface structure*. This distinction is important because a model's deep structure must match the optimizer's capabilities in order for the optimizer to be applicable, whereas the same is not true of surface structure. That is, deep structure would be fixed in each delivery system, but surface structure would be flexible. Deep structure impacts the optimizer engine, but surface structure does not.

To illustrate the deep structure versus surface structure distinction, we will use the classical, single period, single product transportation problem with one twist: some of the customers are sole-sourced.

The deep structure is given by the minimal symbolic mathematical structure needed to represent this situation in a way that is data-free and dimension-independent. Here it is, stated in terms of model elements:

1. A list of plants, indexed by i, and their (upper) capacities C_i (units).
2. A list of customers, indexed by j, and their demands D_j (units). Demands must be met or exceeded.
3. A list of sole-sourced customers that is a subset of the list of all customers.
4. A list of transportation links between plants and customers, a unit transportation cost T_{ij} (\$/unit) for each, and a planned transportation flow X_{ij} (units) for each.

As usual, the aim is to find the transportation flows X_{ij} which minimize total transportation cost subject to staying within the plant capacities, meeting all customer demands, and sole-sourcing the designated customers. Assume that this problem must be solved just prior to each planning period, which might be a week in duration (the model itself is static, but the data change weekly).

Optimizers exist that can solve this problem efficiently. For any such optimizer, an input file can be structured for this class of problems once and for all. It is a simple matter to stuff the numbers into the appropriate places for any particular problem instance.

So much for deep structure. What about surface structure? That depends on how the numbers are to be generated for any particular application. Any "front end submodel" that computes the C_i's, the D_j's, and the T_{ij}'s —or even that computes the lists of plants, customers, and links— can be viewed as surface structure. And any "back end submodel" that summarizes the optimal X_{ij}'s and/or model data without further optimization can be viewed as surface structure.

There are many different possibilities for surface structure. We will consider three pertaining to the front end and two pertaining to the back end.

Capacities

The capacities C_i are calculated as the minimum of three different kinds of capacity: machine capacity, raw material capacity, and labor capacity. All are measured in units during the planning period.

Machine capacity MC_i depends on the production lines at plant i and their maintenance state. The maintenance states, indexed by m, are: 0 (down), 1 (needs major maintenance), 2 (needs minor maintenance), and 3 (just fine). Let p index the production lines (in all plants). Let the installation plan be given by the index sets $P(i)$ giving the indices of the production lines at plant i. Let $M(p)$ be the current maintenance state of production line p. Let R_{pm} be the conditional production capacity (units) of line p when it is in maintenance state m. Thus

$$MC_i = \sum_{p \in P(i)} R_{pM(p)}.$$

Raw material capacity RC_i reflects the fact that raw material shortages can limit what can be produced in the planning period. There is only one raw material. The company likes to express raw material availability at plant i as A_i, the percentage of what is needed to produce up to machine capacity under the supposition that all machines are at the highest maintenance level. Thus

$$RC_i = \frac{A_i}{100} \sum_{p \in P(i)} R_{p3}.$$

Labor capacity LC_i arises because the finishing step at the end of the production line is largely manual. Labor productivity L_i (units per labor hour) varies by plant. There are H_i labor hours available in the planning period at plant i. So

$$LC_i = L_i H_i.$$

Finally, we have $C_i = \min\{MC_i, RC_i, LC_i\}$.

Demands

Demand estimates come from a simple one-ahead exponential smoothing forecast applied to past data. Thus, the D_j's are calculated from past historical demands, $HD_{j1}, HD_{j2}, \ldots, HD_{jT}$, using a smoothing constant α. That is,

$$D_j = E_{jt} + \frac{S_{jt}}{\alpha} \quad \text{for } t > 1$$

where

$$\begin{aligned} E_{jt} &= \alpha HD_{jt} + (1-\alpha)E_{jt-1} && \text{for } t > 1 \\ &= HD_{j1} && \text{for } t = 1 \end{aligned}$$

and

$$\begin{aligned} S_{jt} &= \alpha(E_{jt} - E_{jt-1}) + (1-\alpha)S_{jt-1} && \text{for } t > 2 \\ &= E_{jt} - E_{jt-1} && \text{for } t = 2. \end{aligned}$$

Production Costs

Production costs PC_i (\$/unit), which vary by plant and include finishing labor, are to be taken into consideration so that the model will be more useful for plant loading decisions. Of course, production costs can be added to the unit transportation cost so far as the optimizer engine is concerned, but these two cost components need to be developed and reported separately.

Production Report

Based on the optimal solution X^*, production by plant and associated production costs must be reported. Thus

$$\sum_j X^*_{ij} \qquad \text{(units)}$$

and

$$PC_i \sum_j X^*_{ij} \qquad (\$)$$

must be calculated and reported for each i.

Finishing Labor Report

Based on the optimal solution X^*, finishing labor by plant must be reported. Thus

$$\sum_j \frac{X^*_{ij}}{L_i}$$

must be calculated and reported for each i.

It would be easy to think up many other possible surface structure features for the above example. Different clients will want different combinations of these and other surface structure features. There is no effective limit on this sort of thing. Infinite surface structure variations are possible on any deep structure theme, even for a nearly trivial one like that considered here. The result: a single optimizer engine might have to be delivered in a wide variety of superficially different application packages.

Table 1: Volatility versus Intrinsic Ease of Change

Aspect of a model	*Degree of volatility*	*Intrinsic ease of change*
A. Data value	high	very easy
B. Index set content	moderately high	easy
C. Surface structure	moderately low	not easy
D. Deep structure	low	difficult

2.2 Principle of Graduated Flexibility

Another requirement is that a good delivery system should employ a model representation and implementation design that obeys what might be called the *Principle of Graduated Flexibility*:

> A model's mathematical structure and detailed data should be represented and implemented in such a way that, over the lifetime of an application, the most commonly required model changes are the easiest to make, while less commonly required changes may be more difficult.

This Principle is simple economics applied to model-based work, and derives from objectives 2, 3, and 4.

It is instructive to apply this Principle to changes having to do with data (coefficient values and index set content) and with surface and deep mathematical structure. Table 1 notes the typical occurrence frequency (volatility) and intrinsic ease of making such changes.

It is lucky that volatility and intrinsic ease are so highly correlated for these important model aspects. "Intrinsic" ease refers to the direct effort needed to make a change assuming an ideal model representation and implementation. In reality, the actual ease of making a change can be worse than the intrinsic ease. This occurs when a model's representation or implementation are such that making a change in one aspect of a model turns out to impact the implemented representation of another aspect. This type of inefficiency-creating interaction typically occurs for one or more of the following reasons:

1. data are commingled with mathematical structure,
2. certain inappropriate kinds of indexing structures are used,
3. the surface and deep aspects of mathematical structure are excessively intertwined.

Any of these could be the fault of either the representation or its implementation.

It follows that the Principle of Graduated Flexibility requires model representation and implementation to be designed so that changing aspect A does not impact aspects B or C or D, changing aspect B does not impact aspects C or D, and changing aspect C does not impact aspect D. In other words, none of the natural dependencies — A depends on B and C and D, B depends on C and D, C depends on D — may be inverted.

2.3 Central Role of a Modeling Language

A third requirement for a good delivery system is that a modeling language should play a central role in its architecture. This requirement holds for all delivery systems oriented toward optimization. The reasons are made clear by R. Fourer's classic paper [7]: user productivity, model quality, maintainability, evolvability, and reducing the number and level of skills needed to make optimization accessible to practitioners.

The design and implementation of new modeling languages is a very active area at present, and it is fortunate that good facilitating tools from computer science are available. (See [8] and [9] for discussions on the design of modeling languages and [10] for a survey on model management.) It turns out that these tools are needed to satisfy the previously discussed Principle.

It is important to note that most modeling languages are supported by a modeling language translator that converts the modeler representation into a solver representation. In that sense, a translator is similar to a computer programming language interpreter. For a good delivery system, however, a program that generates a problem generator, which is an executable program that will take the data and convert it into the format required by the solver, is needed or else the problem generator has to be custom coded. In that sense, the generator is closer to a computer programming language compiler.

A generator tends to be more difficult to implement than a translator. However, if properly implemented, it can be much more efficient, both in terms of speed and the amount of memory required, since it does not have to analyze the complete model structure each time it solves a problem, as the translator does. If the model structure changes, however, generating a new problem generator will probably take longer than using a translator. In the case of a delivery system, the generator approach should be much more efficient

since the model structure will not change, although the data will.

3 Design of an Optimization-Based Decision Support System

In order for an optimization-based DSS to be used, it is crucial that its design adequately serves the needs of the user. This design, in turn, will have a strong impact on our approach for developing these systems. Since the design of an optimization-based DSS tends to be evolutionary, we will implement some systems for the participating manufacturing firms during the first year of the project. This will give us some experience with which to refine our approach.

The architecture of the optimization-based DSS being implemented has six distinct parts, as shown in Figure 1: a user interface, a scenario manager, a database management system, a problem generator, a solution interpreter, and a solver. The user interface is being built largely by using new graphical user interface generation tools and will interface with some commercial relational database management system. The solver to be used, on the other hand, will depend on the type of deep structure of the problem. Given a deep structure, however, the solver will be fixed and in most cases a commercial code, such as CPLEX or OSL, will be used.

The problem generator has the task of generating the solver representation used by the solver. It obtains the data it needs directly from the database management system in order to allow easier customization. The solution interpreter simply has to convert the solver's output to a database format so it can be manipulated by the database management system.

Most of the interaction between the user and the system is through the DBMS, which makes it a crucial element in the system. Fortunately, many very good commercial database systems now exist and can provide many of capabilities required by the system, such as report generation for example. Finally, the scenario manager would help keep track of the different sets of input data and the corresponding sets of output data that are generated during the solution of most problems.

Three of the key design decisions of the architecture are: how much the user can customize the system, how to build the problem generator, and how to interact with the solver. We discuss each of these.

3.1 User Customization

This issue relates to the Principle of Graduated Flexibility. The question is: how much can the user change the model. One alternative is to allow the user to modify data values, index set content, and surface structure. Another alternative would be to only allow the user to modify data values and index set content.

The main advantage of the first alternative is that the user could make many changes to the DSS, making evolution easier. The second alternative, on the other hand, would result in simpler, smaller, but much less flexible DSS. Furthermore, the first alternative would require a more complex user interface in order to modify the surface structure of the model and there would be a greater risk of an inexpert user introducing errors into the model representation.

For most existing modeling languages, the first alternative is very difficult to implement because of the implications of certain changes on the way data is handled. It could require, for example, reading the data in an entirely different way and having to change how the values of certain entities are calculated.

3.2 Problem Generator

One possible approach for the design of the problem generator is that used by most modeling languages, namely using a translator. In this approach, the problem generator would read the specification of the model in a suitable modeling language, and the data from the database, and then generate the problem in the format required by the solver. This is the approach used by the FW/SM-MPS Interface, of the FW/SM prototype [11, 12]. The advantage of this approach is its generality. It can easily accommodate almost any change in the model representation, so long as it does not require changing the solver. The disadvantage is that it tends to be slow and inefficient, since the model has to be translated every time the problem is solved.

A different approach is to exploit the deep structure versus surface structure distinction, discussed earlier, by having a problem generator program supporting a given deep structure, and handling the surface structure with different tools. This approach should result in a faster and more efficient optimization-based decision support system, since the model would not have to be translated each time the problem is solved. However, a program would have to be written, or automatically generated, for each type of deep structure, that efficiently generates the solver representation for

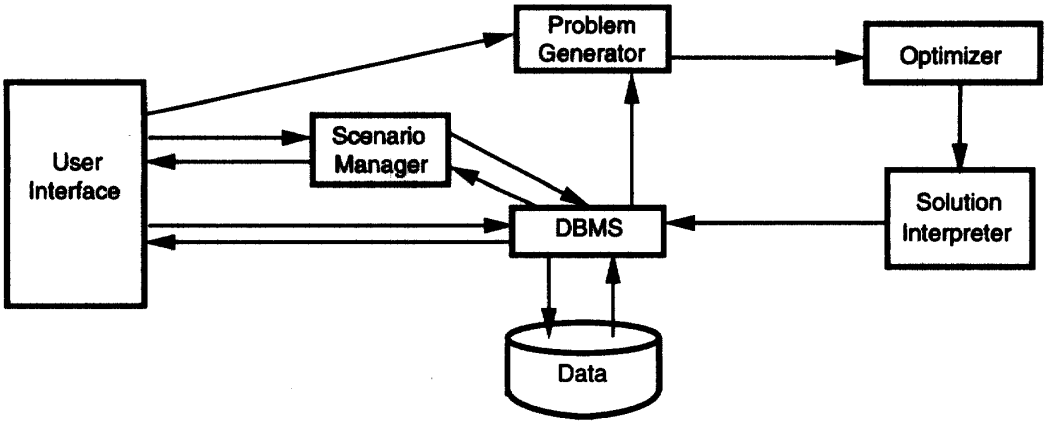

Figure 1: Main Components of an Optimization-based Decision Support System

that deep structure, making this approach harder to implement.

Another difficulty is that some other tool, like another compiler, is needed to support the surface structure. Furthermore, in some cases, the deep structure of the problem could depend on the data. For example, for certain data a cost element might be constant, making it a linear programming problem, and for other data, it might be a linear function, making it a nonlinear programming problem. There are at least three alternatives that can be considered:

- disallow this type of model representation
- support both types of deep structures, by having a DSS with the ability of using either a linear or nonlinear solver, depending on the data, or
- use the more general representation, in this case, always use a nonlinear solver, although the problem would be linear in many cases.

3.3 Solver Integration

Solvers can be either loosely or tightly integrated with the rest of the system. Loose integration means that the interaction between the solver and the rest of the system is by reading and writing text files. Tight integration means that the interaction between the solver and the rest of the system is by sharing some data structures in the computer's primary storage (RAM). (For a good discussion of solver integration see [13].)

Loose integration is much simpler to implement, but tends to be less efficient since the solver representation has to be first written, and then read from secondary storage (hard disk), which is usually slower than all-in-memory processing. Furthermore, the solution then has to be written by the solver to the disk and then read back in.

Tight integration is harder to implement, since it is more solver-dependent and difficult to debug, but tends to be much faster, since it requires much less writing and reading on secondary storage.

Tight integration also restricts the number of solvers that can be used since either the source code or a linkable subroutine is needed, while loose integration only requires an executable version of the solver. On the other hand, tight integration could permit a much closer interaction between the solver and the program generator, which could lead to faster and more robust solutions. Furthermore, in the Windows environment, there is an increasing number of solvers available as dynamic link libraries (DLLs), which can be used to provide tight integration with the solver.

3.4 Discussion

Since most of the issues involved in the design of the optimization-based DSS are empirical, we will experiment with the different alternatives to determine which ones seem to work better. This will also give us the chance to determine what kind of user interface is the most effective for each kind of problem.

In fact, more than one user interface may be required for a given problem, since, in most cases, more

than one user uses the system. For example, one user may be in charge of updating the data, while another may be in charge of running the system to support the decision making process.

There had been an earlier effort to implement a production planning DSS for one of the firms participating in the project. Although the model developed in this case was adequate, the user interface and the solver were considered poor, which helped explain why it was not being used, despite having been developed very recently. This highlights the importance of having a well designed and properly implemented optimization-based DSS.

4 Proposed Approach

The objectives and requirements stated previously for good optimization-based application packages could be met through different approaches. Assume for the sake of discussion that a particular optimizer has been chosen. It could be a general purpose optimizer or one that is highly specialized.

How can one inexpensively build so many different application packages for a single optimizer? The answer is that tools are needed to largely automate the customization:

1. The first requirement is to have a *language* in which surface structure can be described.

2. The second requirement is to have *processors* for this language that can check surface structure descriptions for consistency, can structure data tables, can calculate the final coefficients for optimization (e.g., the S_i's and D_j's and T_{ij}'s of the extended transportation example) without having to write any code, and can do the back end calculations without having to write any code.

3. The third requirement is to have a *translator* or problem generator that can prepare the optimizer's input file automatically.

4. The fourth requirement is to have tools for generating state of the art *graphical user interfaces* and connections to relational database management systems.

4.1 Surface Structure Language

A suitable language and its processors are already available: the SML language (see [14], [15] and [16]) and the programs SMLCheck, Interp_Ck, EDGEN, and FcEval from the FW/SM prototype [11, 12].

SML has ample expressive power, completely specified syntax and semantics, a documentary sublanguage whose consistency can be checked against the mathematics, hierarchical features to help manage complexity, and total structure/data independence. Moreover, SML seems to lend itself to the surface/deep structure distinction; basically by declaring certain paragraphs of a given model to embody deep structure and protecting them from most types of editing. Furthermore, as noted in [12], the explicit database scheme mapping the SML schema to instantiating data, can be used by the processors to quickly locate data and improve the interaction with database management systems.

It is important to note that many of the notable characteristics of SML mentioned in [15], besides those mentioned above, are also particularly relevant in the context of generating optimization-based DSSs. For example, the meticulous attention to the explicit specification of definitional dependencies; the sublanguage for making documentary comments that has nearly all the permissiveness of natural language, yet can be checked for semantic consistency; and the comprehensive specification of semantics as well as syntax for both general model structure and model instance can help make the process of developing and maintaining the models more manageable.

4.2 Processors

A number of available software modules from SML's FW/SM implementation (see [12]) can be used to provide extensive error-checking, automatic production of reference documentation, automatic production of demonstrably good data table designs, and even a compiler that produces C code enabling optimizer-independent immediate calculation of all model-related quantities (much as one can do in spreadsheet programs). The latter capability is, to our knowledge, unheard of in delivery systems for optimizers, and should be very useful for debugging, "What If" studies, and other kinds of model and results analysis. These programs can calculate the final coefficients needed for optimization for any reasonable front end surface structure, and can do the back end calculations for any reasonable surface structure, all without having to write any code! This C program does, however, require adaptation to the chosen database package.

Detailed data would be organized into tables (relations) whose layout is determined by mathemati-

cal structure, but which do not commingle with the representation of mathematical structure. Nor would data values and index sets be commingled. Besides conferring all the advantages of structure/data independence and helping compliance with the Principle of Graduated Flexibility, this would enable modern commercial database technology to be used to manage the large volumes of data that arise in most practical applications. This last point may prove to be critical: the integration of commercial database technology with an optimizer engine would contribute greatly to all four objectives.

4.3 Translator

Initially a translator will be custom coded, but this need only be done once for any particular deep structure no matter what surface features are desired. Achieving representational independence between deep and surface structures is one of the keys to making this task a one-time affair. The other key is having a compiler (FcEval in this case) from the surface structure language (SML in this case) to generate the code to do all surface structure calculations.

Pushing the optimization button would cause the following to happen in a way that is completely transparent to the user.

1. FcEval would calculate the final coefficients needed for optimization, and put the results into the proper client-accessible tables. (These tables would NOT be client/installation-specific, although other tables catering to surface structure would be.)

2. The system's standard translator (which is NOT client/installation-specific) would build the optimizer's input file using the calculated final coefficients.

3. The optimizer would be invoked.

4. The optimal solution would be loaded into the proper client-accessible table. (This table would NOT be client/installation-specific.)

5. FcEval would do the back end surface structure calculations and put the results into the proper client-accessible tables. (These tables WOULD be client/installation-specific.)

6. Clients could look in the solution table and the back end tables for their results. If the host database package is capable of producing hot-linked forms or reports, as some database management systems are, then such forms and reports could be produced automatically.

In the second stage of the project, a translator writing system, similar to the one described in [17], will be implemented that will automatically generate the translator, from the SML representation, to make the process even more efficient. Note that the translator writing system would have its own user interface geared to the needs of the modeler, as opposed to those of the end user of the optimization based DSS, as shown in Figure 2.

Another advantage of the translator writing system approach is that the complete system would be generated from a single representation, since both the deep structure and the surface structure would be described in the same language.

4.4 Graphical User Interface

Most users today expect a graphical user interface (GUI) in every program they use. Therefore, we need to generate programs that have a modern-looking GUI. Fortunately, many tools already exist, and many more are being added, that facilitate the implementation of GUIs. In particular, the Microsoft Windows environment has some very good tools which makes it an attractive platform for implementing the proposed system. Most of these tools, like PowerBuilder, SQL Windows, or Visual Basic, have also connections to relational database management systems, which can be used to access the data required by the model.

Furthermore, some of these tools also support a client/server architecture, where the database management system, for example, could be on a different computer than the one running the optimization-based DSS. The program that has been initially chosen for the project, PowerBuilder, has all of these capabilities.

As noted earlier, the complexity of the user interface will be determined by the degree of user customization we will provide with the system. In particular, the problem is how can the user access and modify the surface structure representation.

Another design issue that has not yet been resolved is how to implement a "scenario manager" for the system. It seems that the easiest way of implementing it is by using the capabilities of the database management system. In this way, most of the operations the user would carry out, except invoking the optimizer,

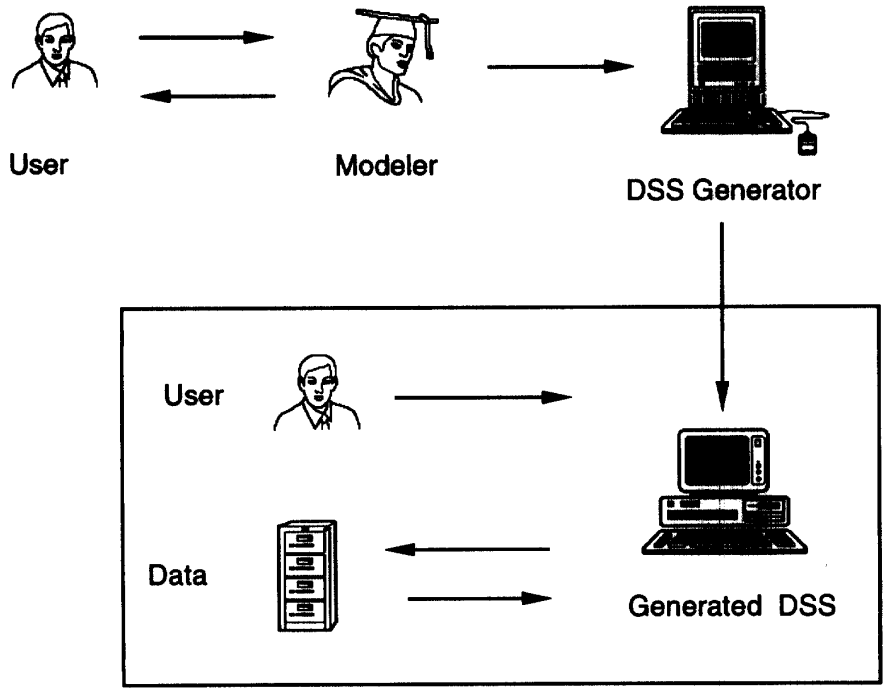

Figure 2: Architecture of a Generator of Optimization-based Decision Support Systems

would be mainly handled by the database management system.

5 Conclusion

An approach has been presented that attempts to make the process of implementing optimization-based decision support systems easier, faster, and less costly than it is today. Instead of trying to replace OR/MS analysts (or modelers) by giving sophisticated tools to the users, to enable them to solve their problems, we suggest giving tools to the modelers to allow them to develop optimization-based DSSs for users without having to write any code.

As a point of departure, we choose to restrict ourselves initially to optimization problems for production, distribution, and inventory. We shall test our approach by building custom prototypes for two Chilean manufacturing firms that are participating in the project. One of the firms makes refrigerators, washing machines, and stoves, while the other makes packaged food.

Once the custom prototypes are working well, and we are convinced that they serve their purpose, we will concentrate on improving the tools for generating similar systems. As mentioned before some of these tools have already been implemented, other tools can be purchased (like PowerBuilder and similar tools), while others have to be implemented. We also need to refine the design of the architecture that will allow us to integrate the different sets of tools and improve or adapt some of the tools.

If our approach works well, we could extend it to different types of problems, and also attempt to generate an application package for a more generic type of problem. These two things combined will provide high quality, application-specific delivery systems for optimizer engines at a reasonable cost, thus helping to realize the full potential of optimization technology.

Acknowledgments

We offer thanks for support provided by FONDECYT (94-0637), FONDEF (2-55), and Shell Development Corp.

References

[1] H. Schultz and W. Pulleyblank, "Trends in Optimization," *OR/MS Today*, vol. 18, pp. 20–25, August 1991.

[2] J. Little, "Operations Research in Industry: New Opportunities in a Changing World," *Operations Research*, vol. 39, pp. 531–542, July–August 1991.

[3] T. Willemain, "Insights on Modeling from a Dozen Experts," *Operations Research*, vol. 42, pp. 213–222, March–April 1994.

[4] S. Gass, "Model World: Danger, Beware the User as a Modeler," *Interfaces*, vol. 20, pp. 60–64, May–June 1990.

[5] A. Sadrian and Y. Yoon, "A Procurement Decision Support System in Business Volume Discount Environments," *Operations Research*, vol. 42, pp. 14–23, January–February 1994.

[6] P. Gazmuri, S. Maturana, F. Vicuña, and L. Contesse, "Diseño de un Generador de Sistemas Computacionales para la Optimización de Procesos Productivos," FONDEF Proposal, 1993. Pontificia Universidad Católica de Chile.

[7] R. Fourer, "Modeling Languages Versus Matrix Generators for Linear Programming," *ACM Transactions on Mathematical Software*, vol. 9, pp. 143–183, June 1983.

[8] C. Kuip, "Algebraic Languages for Mathematical Programming," *European Journal of Operational Research*, vol. 67, pp. 25–51, 1993.

[9] S. Maturana, "Issues in the Design of Modeling Languages for Mathematical Programming," *European Journal of Operational Research*, vol. 72, pp. 243–261, 1994.

[10] R. Krishnan, "Model Management: Survey, Future Research Directions and a Bibliography," *ORSA CSTS Newsletter*, vol. 14, pp. 1–22, Spring 1993.

[11] A. Geoffrion, "FW/SM: A Prototype Structured Modeling Environment," *Management Science*, vol. 37, pp. 1513–1538, December 1991.

[12] L. Neustadter, A. Geoffrion, S. Maturana, Y. Tsai, and F. Vicuña, "The Design and Implementation of a Prototype Structured Modeling Environment," *Annals of Operations Research*, vol. 38, pp. 453–484, 1992.

[13] A. Drud, "Interfaces Between Modeling Systems and Solutions Algorithms," in *Mathematical Models for Decision Support*, (G. Mitra, ed.), (Berlin), pp. 187–196, NATO ASI Series, Vol. F48, Springer-Verlag, 1988.

[14] A. Geoffrion, "Introduction to Structured Modeling," *Management Science*, vol. 33, pp. 547–588, May 1987.

[15] A. Geoffrion, "The SML Language for Structured Modeling," *Operations Research*, vol. 40, pp. 38–75, January–February 1992.

[16] A. Geoffrion, "Structured Modeling: Survey and Future Research Directions," *ORSA CSTS Newsletter*, vol. 15, pp. 1–20, Spring 1994.

[17] S. Maturana, *A Translator Writing System for Algebraic Modeling Languages*. PhD thesis, Anderson Graduate School of Management, UCLA, 1990.