Design and Implementation of an Optimization-Based Decision Support System Generator

Sergio Maturana\textsuperscript{1} \hspace{1cm} Juan-Carlos Ferrer\textsuperscript{2} \\
Francisco Barañao \\
Pontificia Universidad Católica de Chile, Casilla 306, Santiago 22, Chile \\

May 1, 2002

\textsuperscript{1}email: smaturan@ing.puc.cl \hspace{1cm} \textsuperscript{2}email: jferrer@ing.puc.cl
Abstract

Decision-makers today need to be able to rapidly find good solutions to increasingly complex problems. Optimization-based decision support systems (OBDSSs) can help decision-makers to meet this challenge. Building such systems, however, is expensive and time-consuming.

In this paper we present the design of an OBDSS generator that greatly automates the development process by integrating a model specification with a database, a solver, and a graphical user interface (GUI). The generator automatically creates the database for storing the instantiating data, the GUI, and the solver interface. With this generator, an OR/MS analyst can build an OBDSS faster and more efficiently than using traditional tools. The resulting OBDSSs can be used by decision-makers with little knowledge of the underlying OR/MS technology. We also discuss our experience using the generator to develop production planning and truck dispatching OBDSSs for several firms in the context of a 3-year R&D project.

Keywords: Decision Support Systems, Modelling Languages, Optimisation.
1 Introduction

Over the years, the Operations Research/Management Science (OR/MS) field has developed a wide variety of tools to help solve real problems in different areas. As noted by Dutta [1], mathematical models, particularly optimization models, have been, and continue to be, the workhorses for decision support. However, when practitioners try to use OR/MS tools in different business situations, they frequently have to overcome many obstacles. Many of these obstacles would be easier to overcome with better tools.

The tools required to support OR/MS analysts in their model-based work depend on the situation they have to face. When the model is used to solve a problem only a few times—for example, when deciding the optimal location of a plant—the analyst can use specialized tools to model and solve the problem and present the solution to the decision-makers. When the model is used to support a decision that is made repetitively—for example, when deciding how to distribute the products of a company to a large number of clients in different locations—the analyst, after having developed the model, usually does not continue supporting the decision-maker, who must then interact directly with the underlying OR/MS tools.

For the first type of model-based work, modeling systems are well suited. Today there are many systems that use a modeling language to specify the problem to be solved. These languages can be either graphical or textual; procedural or declarative; general or specialized [2]. Most modeling systems use an algebraic modeling language [3], such as AIMMS, AMPL, GAMS, LINGO, LPL, MGG, MODLER, MPL, and XPRESS-MP. Other approaches have also been proposed, such as an environment that integrates a hypertext system with a graphical tool for creating and solving linear programming models [4]; and a mixed-integer programming (MIP) system based on a relational database management system (RDBMS), which uses the block schematic approach and SQL, the de facto standard language for data manipulation [5].
In order to use a modeling system, a person must know the system’s modeling language and have some notion on how the solver works, besides the knowledge related to the problem itself. Although this level of knowledge is a reasonable requirement for an analyst, it is not for many decision makers. In a sense, the analyst acts as an intermediary between the decision maker and an OR/MS application. An alternative approach proposed by some authors (see [6] and [7]), is to provide tools that can allow non-expert users to construct their own models and use these models to support their decisions. Some examples of this approach are MODFORM [8] and AEROBA [9].

The second type of model-based work, which supports repetitive decisions, requires different tools. In this case it is necessary to bundle the OR/MS technology in a system that can support a decision-maker who may not have an OR/MS background. This suggests the need for optimization-based decision support systems (OBDSSs), which integrate the solver, database management procedures, and a user interface into a package that appears to the user as a single application supporting the decision-making process for a certain problem [10]. OBDSSs help to make optimization technology available to a larger number of users by allowing production planners, warehouse managers, and other decision makers, to use optimization tools without the direct intervention of an OR/MS analyst.

Other authors [11] also note the need of bringing solver, model, and data together within the same application. They stress the importance of having what they call an Integrated DSS, which can support different types of users, such as solver experts, modeling experts, database experts, domain experts, and decision makers. In our terminology, an OBDSS would be the part of this system designed to support the decision makers.

The objective of our work was to develop an OBDSS generator, which is a set of tools to support the design and construction of an OBDSS. This tool is especially designed for the second type of model-based work mentioned above, so we are assuming that there will be an OR/MS analyst,
which we will call an implementor, and a decision maker, which we will call the user, who may have no previous knowledge of OR/MS tools. The OBDSS generator is intended to be used by the OR/MS analyst to generate an OBDSS, which can then be used by the decision maker, without further assistance of the analyst. Therefore, we will be referring to two types of systems: the generator, which is used by the implementor (analyst), and the OBDSS, which is used by the user (decision-maker). By using the generator, an implementor can develop an OBDSS in a fraction of the time and the cost it would take using a traditional approach.

The approach we used to develop the OBDSS generator, integrates recent advances of both OR/MS and computer science. From OR/MS we borrowed structured modeling [12] and optimization algorithms. From computer science we borrowed relational database technology, modern GUIs (graphical user interfaces), and software engineering techniques. This work was carried out in the context of a 3-year R&D project called GESCOPP (described in Spanish in [13]), which has produced several prototypes in different areas. One of these is described in [14].

The remainder of this article is organized as follows. In Section 2 we discuss some approaches that are related to the process of building OBDSSs. In Section 3 we discuss the design of the OBDSS generator. In Section 4 we illustrate the features of the generator by showing how it can be used to generate an OBDSS. In Section 5 we discuss advantages and disadvantages of using the generator. We end, in Section 6, with some concluding remarks.

2 Building Optimization-Based DSSs

The process of designing, constructing, and implementing a DSS is complex. It involves issues ranging from technical (such as hardware selection and networking) to behavioral (such as person-machine interfaces and the potential impact on individuals and groups) [15]. Building and implementing OBDSSs is even more complicated since it also involves developing a mathematical model
Several different approaches have been proposed to support the process of building OBDSS. Many of these have centered on the modeling stage, while others have proposed tools to support the development of OBDSSs. Some have been restricted to certain types of problems, others have proposed architectures to facilitate the integration of models, data and algorithms, while others have proposed the use of DSS generators. We will briefly discuss some of these approaches.

Makowski [16] discusses many of the issues involved in the design and implementation of model-based DSSs, in particular those that use simulation and optimization to analyze a model. He notes that the requirements for data handling and user interface often make it impractical to use existing general purpose modeling tools, especially for many complex, real-life applications. He also states that when a certain type of DSS will be generated many times, it may be sensible to develop specialized tools to help generate it.

Much research has been conducted on the process of developing the model of a DSS. Muhanna [17] proposed an object-oriented framework for model management and DSS development, which integrates Geórfion’s structured modeling approach [18] with Muhanna and Pick’s systems approach [19]. Kwon and Park [20] proposed a prototype called RMT based on the concept of “reverse” modeling. Contrary to the typical modeling life-cycle, reverse modeling focuses on the identification, extraction, and reorganization of model constructs from existing models. Pillutla and Nag [21] stress the importance of rapid and automated model development for decision support in production scheduling applications, where problem instances are often similar to some general model, but not necessarily consistent with it. They develop a scheme to represent model information in an object-oriented framework that relies on the definition of natural entities.

Desrochers et al. [22] present a different approach, which emphasizes the modeling and solution of physical distribution systems. They propose a system that incorporates the current knowledge
and expertise related to solving vehicle routing problems (VRPs), which are notoriously difficult to solve using standard mathematical programming tools.

Holocher et al. [23] present an architecture for model-oriented integration of data and algorithms (MIDA), which can be used for building distributed DSSs, integrating high-level modeling approaches and implementation techniques for platform independent distributed client/server applications. They implemented AMBAS-SOOM, a variant of the MIDA architecture, which uses AMBAS as a distributed platform and SOOM as the modeling environment.

Bhargava et al. [24] evaluated DSS Generator software in the realm of decision analysis methods. They define DSS generator as a kit or environment for developing an application-specific DSS. Rico et al. [25] present a prototype DSS Generator, which can generate DSSs for different types of problems. The generator includes a set of standard prefabricated decision models, yet it also allows users to add their own models. The DSS Development System (DSSDS) [26], also speeds up the development and modification of a DSS by using prefabricated components and highly automated tools throughout the development process.

Finally, it is important to mention a particular type of generator, called translator writing systems or application generators [27], which translate high-level specifications to low level code. The high-level specifications can take the form of an interactive dialog, graphics, or text. This type of system has been used to build compilers for many different programming languages.

3 The OBDSS Generator

In this section we discuss the design of the OBDSS, which will be generated by the generator, and the general design of the generator.
3.1 Design of an OBDSS

Before designing the generator, we designed the OBDSS that it would generate so that it would have easily reusable components. Of the five parts that comprise an OBDSS (Figure 1), our design requires generating three of them: the user interface, the database structure (data management), and the model structure. We briefly discuss the design of the components of the OBDSS that is constructed with the help of the OBDSS generator we developed.

The user interface is the most visible of the components of an OBDSS. It allows the user to manipulate the data of the problem, solve it, and analyze the results. It also provides access to tools for storing different instances of the problem and for managing many scenarios. Until now, OBDSS developers have been rather slow to react to the growing importance of graphical user interfaces. Rathnam and Mannino [28] note there has not been a concerted attempt to focus on high-level tools for developing the interface component of a DSS. In the case of an OBDSS, the main challenge of the design of the user interface is to allow a decision-maker with little mathematical background to solve problems using sophisticated tools, such as mathematical programming solvers.

The interfaces of the OBDSSs were implemented using PowerBuilder, which is a high-level tool for developing Windows applications. For a detailed discussion of the design of the user interface
see [29].

The second component of the OBDSS is the model structure of the problem to be solved. Note that we will use model structure and general structure of the model, which is the term used in structured modeling, as synonymous. Structured modeling uses different representations for the general structure and for the detailed data that defines a model instance. We chose SML [30] — a modeling language that supports structured modeling— to specify the model structure since it supports complete independence of the general structure of a class of models from the instantiating data [31]. This is important for our approach since the model structure is fixed for a given OBDSS; that is, the users cannot modify the model structure; they can only modify the instantiating data. Thus, the model structure should be as general as possible. For example, it is important to avoid “hard-wiring” numerical data in the model structure.

The third component of the OBDSS is the data management subsystem. It includes a database for storing all the detailed data of the model and a relational database management system (RDBMS) to manage this data. A given model instance can be obtained by combining this detailed data with the model structure. Fourer [32] notes that in the design and use of large-scale mathematical programming systems, a substantial portion of the effort has no direct relation to the variables and constraints, but is concerned with the description, manipulation, and display of data. He stresses the importance of deriving fundamental principles of database construction for the specific case of large-scale mathematical programming. The RDBMS can access, with a client/server approach, a wide variety of data sources. It also provides excellent facilities for viewing and editing both input and output data, which is one of the most time-consuming tasks involved in solving real problems.

Whenever the user chooses to solve an instance of the model, the instance generator, which is the fourth component of the OBDSS, accesses the database containing the detailed data, reads the
model structure, and generates the input .le for the solver. Currently our OBDSSs are restricted to MIP problems. This is due to the availability of a de facto standard for this type of problem, namely the MPS format.

After the instance generator has generated the input .le, the solver—the .th component of the generated OBDSS—reads the input, solves the problem, and outputs the solution, if it is able to find one. Should the need arise, it would be relatively easy to change CPLEX, the solver we use, to any other LP solver that can read MPS-format .les. However, since there is no standard output format for LP solvers, changing the solver would require modifications in the procedure for reading the solution and importing it into the database.

3.2 Design of the OBDSS Generator

The OBDSS Generator we developed has two main functions: (1) support the modeling phase using SML as its modeling language, and (2) generate a stand-alone OBDSS to be delivered to the final user (or decision maker). The WIN/SM subsystem supports the modeling phase, whereas the application generator subsystem generates the OBDSS. They interact through the model structure and the specification of the database structure as shown in Figure 2.

WIN/SM, which is based on an earlier DOS-based prototype called FW/SM [33], has a special-
ized editor for specifying the model in the SML modeling language; it also has tools for managing data, and an interface to a commercial solver. The SMLCheck program [34], which was part of the FW/SM system, performs extensive error-checking of the model structure, and generates the Internal Representation Tables (IRT) files. The IRT files are a computer-friendly—but hard to read by humans—representation of the model structure representation, which is used by other processes.

The model structure (model schema in SML) must be supplemented by an auxiliary file that specifies which element is the objective function, which elements are the constraints, and whether the objective function must be maximized or minimized. Structured modeling makes a strict distinction between the model and the tasks that can be performed on the model, such as evaluation or optimization. Hence, optimization-related information is not allowed in the model structure. Since we adopted the convention that all test elements are assumed to be constraints, unless otherwise specified, in most of our models we only needed to specify which was the objective function and whether it was to be maximized or minimized.

Once the model structure has been specified, EDGEN [33], which is part of WIN/SM, produces the database structure specification. This is possible because SML has rules for deriving, from the general model structure, a relational database structure design for detailed data that is robust with respect to insertion/deletion/update anomalies, a quality that reduces the burdens of model design, promotes standardization, and is compatible with relational database technology [30].

The application generator, which is based on PowerBuilder, is the subsystem that actually generates the OBDSS. The main difference between an application generator and a traditional compiler, is that the specifications are very high-level and restricted to a particular domain [35]. Application generators are appropriate in domains where many similar systems are developed, a software is modified or rewritten many times during its life-cycle, or many prototypes are needed
The two components of the OBDSS that are not generated are the solver and the instance generator. The solver is a commercial package (CPLEX), which is capable of reading MPS-format .les and solving MIP problems. The instance generator is based on the MPS Interface process, which is also part of the FW/SM prototype. The instance generator reads the data it needs from the database and the general structure, using embedded SQL commands, and produces the input for the solver. This input consists of two parts: a description of the model and its data in the MPS format; and a table linking the variables in the MPS format description to the original names in the SML model. This is required to translate back the solution generated by the solver into the
Another component of the OBDSS, which is not shown in Figure 3, is the RDBMS. This is a very important component since it manages all the data in the OBDSS, including the model structure, which is also loaded into the database. We used a commercial package (Sybase SQL Anywhere), although, with little effort, it would be possible to switch to a different RDBMS. For more details see [34].

4 Generating an Application

In order to illustrate the features of the generator, we will use an example to show how an implementor could use the OBDSS generator to build an OBDSS.

The classical feedmix model, or optimal diet, has been frequently used in the literature to illustrate certain features of modeling languages. For our example, we will suppose that we wish to generate an OBDSS to help us determine a diet that satisfies all the nutritional requirements of certain animals at a minimum cost.

To generate the OBDSS for this example, the implementor must specify the model and setup the project; create the database; define and generate the DataWindows; construct the reports and online help options; and generate the main menu and the executable file for the application. We describe next each of these steps.

4.1 Model Specification

To begin the process, the implementor must specify the model structure using WIN/SM’s specialized editor. Once the model structure is completely and correctly specified, that is, when SMLCheck detects no errors, WIN/SM can display it as shown in Figure 4. This representation can be used to document the model structure. WIN/SM also provides a graphical representation of the model structure, which is easier to manipulate.
The implementor must also specify, in a separate dialog, that the objective function is TOT-COST and that it should be minimized. It is also convenient to enter some information about the model for documentation purposes. For real applications, model documentation is very important to keep track of the different versions of the models.

4.2 Database Creation

The database is the most important element of the generated system. The application generator, as shown in Figure 3, uses the model structure and the database structure specification —both generated by WIN/SM— to generate the database structure.

The generator reads the model structure as a set of IRT files. The database structure specification generated by WIN/SM for the feedmix example is shown in Figure 5. In structured modeling there is a direct correspondence between this representation and the elemental detail table structure shown in Figure 6. The detailed data, or tuples, shown in these tables, correspond to a particular instance of the problem and should be specified by the decision maker using the generated system.
Figure 5: EDGEN Output for the Feedmix Example

<table>
<thead>
<tr>
<th>NAME</th>
<th>STRUCTURE</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUTR</td>
<td>NUTR jj INTERP</td>
<td>1</td>
</tr>
<tr>
<td>MIN</td>
<td>NUTR jj MIN</td>
<td>2</td>
</tr>
<tr>
<td>MATERIAL</td>
<td>MATERIAL jj INTERP</td>
<td>1</td>
</tr>
<tr>
<td>UCOST</td>
<td>MATERIAL jj UCOST</td>
<td>2</td>
</tr>
<tr>
<td>ANALYSIS</td>
<td>NUTR MATERIAL jj ANALYSIS</td>
<td>2</td>
</tr>
<tr>
<td>Q</td>
<td>MATERIAL jj Q</td>
<td>2</td>
</tr>
<tr>
<td>NLEVEL</td>
<td>NUTR jj NLEVEL</td>
<td>2</td>
</tr>
<tr>
<td>TNLEVEL</td>
<td>NUTR jj TNLEVEL</td>
<td>2</td>
</tr>
<tr>
<td>TOTCOST</td>
<td>jj TOTCOST</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 6: Elemental Detail Tables of the Feedmix Example

<table>
<thead>
<tr>
<th>NLEVEL</th>
<th>TNLEVEL</th>
<th>TOTCOST</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>29.94</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>7.71</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P       #TRUE #TRUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C       #TRUE #TRUE</td>
<td></td>
</tr>
<tr>
<td>TOTCOST</td>
<td></td>
<td>7.53</td>
</tr>
</tbody>
</table>
The application generator creates the database structure by generating SQL statements that must be executed by the RDBMS. These statements may be modified by hand, if needed, before running them on the RDBMS. Implementors can change some of the defaults, such as the length of the strings, or the numeric format of the data. They can also specify which tables should be joined before creating the database.

The application generator outputs the SQL statements, which create the database structure, in three stages: generation of tables for the data of the problem, generation of the insertion triggers, and generation of the general structure tables. We describe next each of these stages in the context of our example.

Table Generation

In this stage, the system generates the SQL statements that create the tables for all the data regarding a particular instance of the model. There are four types of tables, which correspond to each of the types of elements in SML: primitive entities tables, compound entities tables, attributes tables, and tests and functions tables.

A primitive entity table must exist for each genus of type primitive entity and each tuple corresponds to a primitive entity. Each of these tables has three fields: the sequence number, the identifier of the primitive entity, and the description of the entity. The sequence number is required to impose order on the primitive entities since SML assumes that primitive entities are ordered, whereas relational databases assume tables are unordered. The third field can be used to give a more meaningful interpretation of the primitive entity. In our example, NUTR and MATERIALS (Figure 6) are primitive entities tables.

Compound entities in SML are defined in terms of either primitive entity elements or other compound entities. The tables for compound entities are composed of sets of tuples that are
members of primitive entities tables or other compound entities tables. Although there are no compound entities in our example, we could have defined the \((\text{NUTR, MATERIAL})\) tuples in the \textsc{Analysis} table as a compound entity. In that case, we could have defined it as either the Cartesian product of the \text{NUTR} and \text{MATERIAL} primitive entity tables or as a subset of this Cartesian product, where each of the members has to be listed in the table.

The attribute tables are used for storing values related to primitive and compound entity elements. Some of these attributes are input data — or parameters of the model — whereas others are variable attributes that will be computed by the solver. In our example, the \text{UCOST}, \text{MIN}, and \text{ANALYSIS} tables (Figure 6) correspond to parameters, whereas \(Q\) is a decision variable table.

Function elements are value-bearing elements with values computed using rules. Test elements are a special type of function elements that can have only the (binary) values True or False. Function and test functions need to be computed by the system and cannot be directly modified by the decision-maker. \text{NLEVEL} and \text{TOTCOST} (Figure 6) are examples of function tables, whereas \text{TNLEVEL} is an example of a test table.

For each table it creates, the application generator must define the set of \texttt{elds}, the primary key, and the foreign keys, if they exist. Each \texttt{eld} of the table must correspond to some genus in the model, except for the sequence and interpretation \texttt{elds} of the primitive entity tables. The primary key is the set of \texttt{elds} that uniquely identify each record. In \textit{SML}, all the \texttt{elds} of the primary key are associated, either directly or indirectly, to primitive entity elements. Foreign keys are constraints that associate the tuples of a table with the elements of another table. They are used to guarantee the integrity of the information. They insure that any change of the elements in the primitive entity table is consistent with the tables that reference those elements. For example, if a primitive entity element is deleted from the \textsc{Materials} table, say \texttt{std}, then all the other elements in the model that are defined over \texttt{std}, such as \texttt{UCOST(std)}, are also deleted.
CREATE TABLE "NUTR"
(
    SEQ NUMERIC,
    "NUTR" CHAR(40),
    "INTERP" CHAR(100),
    PRIMARY KEY ("NUTR")
);

CREATE TABLE "MIN"
(
    "NUTR" CHAR(40),
    "MIN" DECIMAL(18,2),
    PRIMARY KEY ("NUTR"),
    FOREIGN KEY ("NUTR")
        REFERENCES "NUTR" ("NUTR")
        ON UPDATE CASCADE
        ON DELETE CASCADE
);

Figure 7: SQL statements generated for the Feedmix Example

Figure 7 shows the SQL statements generated by the system that create the NUTR and MIN tables. NUTR is an example of a primitive entity table, whereas MIN is an example of an attribute table that depends on the primitive entities of NUTR. During this process, the implementor only needs to adjust the formats of some fields, such as CHAR(40) or DECIMAL(18,2), and specify the desired joins.

Once the tables have been generated, many of them can be joined in order to reduce the total number of tables. In our example, several joins are possible: NUTR with MIN; MATERIAL with UCOST; NLEVEL with TNLEVEL; etc. The joins are not performed automatically; they must be specified by the implementor.

Finally, the generator must create a special table for those tables that do not require a key; that is, those that have a single element (TOTCOST in our example).

Insertion Triggers Generation

The definition of foreign keys guarantees the integrity of the data in the case of deletion of records in the primitive or compound entity tables. However, the integrity can also be compromised by the insertion of a record in the primitive or compound entity tables, since there could be missing tuples for those tables that reference this primitive entity. In our example, if a primitive element, say "B", is added to the NUTR table, it would be necessary to add two rows in the ANALYSIS
CREATE TRIGGER "T_MATERIAL" AFTER INSERT ON "MATERIAL"
REFERENCING NEW AS new_index
FOR EACH ROW
BEGIN
    INSERT INTO "UCOST" ("MATERIAL")
    SELECT "MATERIAL"
    FROM "MATERIAL"
    WHERE
        "MATERIAL"."MATERIAL" = new_index."MATERIAL";
    INSERT INTO "ANALYSIS" ("NUTR", "MATERIAL")
    SELECT "NUTR", "MATERIAL"
    FROM "NUTR", "MATERIAL"
    WHERE
        "MATERIAL"."MATERIAL" = new_index."MATERIAL";
    INSERT INTO "Q" ("MATERIAL")
    SELECT "MATERIAL"
    FROM "MATERIAL"
    WHERE
        "MATERIAL"."MATERIAL" = new_index."MATERIAL";
END

Figure 8: Trigger Generated for the Feedmix Example

table, with the keys (B, std) and (B, add), in order to complete the Cartesian product.

This insertion can be automated by using triggers, which are procedures carried out by the RDBMS when certain events, such as insertions, occur. Each table must have a trigger for each set of fields that depend on a primitive or compound entity genus. These triggers are automatically executed by the RDBMS immediately after an element is inserted into the table associated with the primitive or compound entity tables. When executed, the triggers insert into the associated tables the result of the Cartesian product between the new primitive or compound entity and the other primitive or compound entities that also appear in the associated tables. In the case of tables that are indexed over a sparse set —in SML a compound entity with a SELECT clause— the trigger will not add any elements; they need to be added by the user. Figure 8 shows a trigger that was generated for the feedmix example that automatically inserts rows into all the tables that depend on MATERIAL whenever a new element is added to this table.
General Structure Tables

These tables are used to store the information related to the model structure, such as the structure of the database, information about the genera of the model, and other information used when solving the model (Minimize T OT COST, in our example). Most of this information comes directly from the IRT files and is used by several components, such as the user interface and the instance generator.

The data in these tables cannot be directly modified, not even by the implementor. If modifications are required, the implementor must go back to WIN/SM to edit the model structure and generate a new version of the system.

4.3 DataWindows Generation

Once the database has been created, the implementor must decide which DataWindows will be generated, and the order in which they will appear in the menu. DataWindows are PowerBuilder objects, which act as an interface to the database for SQL queries. DataWindows can be used to view/edit the result of an SQL query, using all the capabilities of PowerBuilder.

The default is that all tables will have a DataWindow and that they will appear in the same order as in the general structure. If the implementor chooses to specify each DataWindow, the generator provides a menu option for each type of table (primitive entity, compound entity, attribute, test, and function). Implementors can choose between a completely automatic DataWindow generation, or an interactive process, which gives them great control over the appearance of the system. Implementors can decide, for example, which fields to display, the order in which they appear, whether Query by Form (a type of Query by Example [36]) will be used, and many other appearance-related options. They can also change the names of the fields that will be shown to the user. By default the generator uses the key phrases of the model schema (stored in the general structure tables) for
the user interface. Key phrases—the underlined word or words in the model schema—are more meaningful to users than .eld names. In our example, UNIT COST is much easier to understand than UCOST.

4.4 Report Construction and Main Menu Generation

Both the report generation and the online help options must be generated using specialized tools. Reports tend to be very application-specific so we preferred to use the generic tools provided by PowerBuilder. There are also several powerful tools for generating an online help system (we used RoboHelp).

For each generated system there is only one instance of the main menu, which appears in the upper part of the main window of the application. The main menu has nine items: File, Edit, Input, Solve, Output, Reports, Options, Window, and Help.

Generated system manage different model instances as a single entity. Therefore the File option is similar to other programs, such as spreadsheets and word processors, which is easy for the user of the generated systems to understand. The disadvantage is that it tends to create redundancies in the data, with the ensuing problems of space and inconsistencies.

The main menu is composed of two separate parts: one that depends on the model, and one that is invariant from system to system. Only the part that depends on the model, which are the input and output items, needs to be generated. This is done by defining a menu that inherits all the attributes of the .xed part of the menu, incorporating the items for viewing/editing the input and output items, which are generated automatically.

The generated code is in the form of native PowerBuilder code in text files. This code is imported into one of the libraries of the generated system. The part of the menu that is generated allows the user to view/edit the attributes, compound entities, primitive entities, and variable attributes of
the model. Each item of the menu has a name and associated code that opens a window through which the user can view/edit the information. These items are arranged under the input or output items of the main menu, depending on whether the user or the solver assigns the values. The names of the items in the menu, under the input and output menu options of the main menu, are obtained from the model using SML’s key phrases (stored in the general structure tables). For more details on the implementation, see [37].

5 Advantages and Disadvantages of the Generator

During the GESCOPP project, the generator was used to develop several OBDSSs for the participating firms. This gave us the opportunity to evaluate the impact of using the generator on four fundamental aspects: GUI, model maintenance, solution of the problem, and data management. For each of these aspects there are advantages and disadvantages, which are discussed next.

5.1 Graphical User Interface

The advantages of using the generator to create the GUI, were an important reduction in the development time, the use of modern interfaces, and the use of software engineering techniques, which improves the evolvability of the generated systems.

The design of the user interface was based on the following criteria: similarity with commonly used software, separation between those parts of the interface that change when a model changes and those that remain constant, and use of object-oriented programming concepts.

The generated systems have two main types of windows: those that do not change from one model to another, such as basic file management windows, and those that do, such as the windows for handling input and output data. For every window —except those used for managing input and output data—an instance can be created when the program is executed (at run-time). This
allows our systems to adapt more easily to different models than if we fixed the attributes when generating the program.

In the case of windows for managing input and output data, an instance of the corresponding class is created each time an item is selected from the menu. This window is modified at run-time by setting its attributes according to the type of the item that was selected. This approach gives an overall consistency to the appearance of the interface of the generated system. Both input and output data are managed using tabular interfaces, which have proved to be a good way of presenting indexed information [38] and can be generated automatically.

We observed a great reduction in development time when using the generator, as compared to the development time using standard programming languages. Once the model is specified in SML, it is possible to generate the application in a few hours. This also meant an important reduction in the cost of developing an OBDSS.

We also observed that inexperienced users learned faster how to use the generated OBDSSs than earlier prototypes we had developed. We think this was because we used a modern GUI that followed—as much as possible— the conventions of a standard windows application. This also helped to reduce the initial resistance to adopting the system.

The ability of quickly generating an application helped the evolvability of the system since we could develop many versions of the OBDSSs in a short time. The users could then use the system and give us feedback on how to improve it.

The main disadvantage of the generator is that it imposes some constraints on the overall design of the user interface. The interface with the data has to be standard, which meant in our case using a tabular-oriented representation. In general this works well for relational databases. There are some cases, however, in which a different representation could be better. For example, in a truck dispatching system, a graphical representation that exploited the geographical aspects
of the problem would probably be easier to understand by most users. Another disadvantage is the performance penalty that is paid by maintaining certain parts of the system generic.

5.2 Model Maintenance

Model maintenance turned out to be very important in our project. Most of our models had to go through many changes. Many of the changes were due to logical or syntactical errors that were quickly detected and fixed. Other changes were necessary to help reduce solution times. These changes took more time since they required experimenting with the solver and different sets of data. Finally some modifications were due to changes in the problems themselves. This type of change required changing models that had not been modified in several months, making model documentation very important.

The features of SML provided important advantages for model maintenance. The strict enforcement of the separation between the model structure and the data allowed us to make many changes by simply modifying data in some tables. When changes needed to be made to the model structure, the automatically generated documentation of SML, and its comprehensive error-checking facilities, proved to be very helpful.

The use of libraries of models as a way of reusing models or parts of a model was also very helpful when developing the .rst version of a model.

The only disadvantage for model maintenance was that SML is relatively difficult to learn. However, if SML is going to be used for many different models, as in our case, the time invested in learning SML pays off.
5.3 Solution of the Problem

The solution in the generated systems is stored in the tables corresponding to variable attributes. These tables can be viewed as tables or as graphs, and they can be exported as spreadsheets. However, they can only be modified if users choose an option for setting them to user-selected values. This option allows users to override the optimal solution determined by the model, if they consider it necessary. Whenever this option is chosen, users are warned that, since they are introducing additional constraints to the underlying mathematical model, they may obtain suboptimal or even infeasible solutions. As a reminder, this warning is repeated each time the users try to solve the problem until they turn this option off.

Since both the instance generator and the solver are fixed for all the generated systems, the solution process in all the systems is the same. In our opinion we achieved our objective of making the solution process as transparent to the user as possible. Also, since the solver is a separate part of the generated system, it is possible to quickly change from the current solver, CPLEX, to any solver that can read the MPS-format files.

Although the solution process is transparent to the user, knowledgeable users can change some of the parameters of the solver. The proper settings of these parameters may significantly impact the solution times of some problems, especially in the case of MIP problems. Unfortunately, most users will probably not possess the required knowledge. Even knowledgeable users need to experiment to find the best settings for these options, since there are no general rules that we are aware of.

Finally, the use of the client/server architecture allows us to solve the problem on a more powerful computer, and then import the solution back to the application.

The main disadvantage, in the case of the solution process, is that the architecture is oriented towards MIP problems. Other problem types could also be solved by writing an instance generator for other types of solvers, such as heuristics. This requires more programming, making the
Another disadvantage of the fixed instance generator and solver, is that it makes it difficult to use problem-specific information to improve the solution process. This can be a very important disadvantage if the problem is very large or difficult to solve using standard optimization tools. Also in the case of very large problems, the time required to generate the MPS-format file can be too long. In this case it may be necessary to write a matrix generator, with all the problems it involves. Nevertheless, the modular design of the generated systems makes it relatively easy to change the instance generator and the solver to allow the solution of much larger problems, without affecting the rest of the system.

5.4 Data Management

The main advantages of the use of the generator probably concern data management. The use of a RDBMS to handle all the information of the model, and the close integration with the structure of the model, make it possible for the generated applications to have very powerful and friendly facilities for managing data.

The use of a RDBMS also helps improve the consistency and integrity of the data. The application can handle events, such as adding or deleting primitive entity elements, that can affect the population of other tables derived from these primitive entities. The client/server approach also allows easy access to data stored in remote databases, making the process transparent to the user. This is important for our generated systems since they are designed to run on desktop computers; many times the data needed to solve the problem, may be stored in a corporate computer.

The only disadvantage, with respect to data management, is that we are restricted to the relational model and to RDBMSs. Other types of data models — such as network, hierarchical, or object-oriented — are not currently supported.
6 Conclusion

OBDSSs may be a very effective way of packaging OR/MS technology for certain kinds of problems. This may promote the use of this kind of technology, as it makes it transparent to the user, provided the OBDSSs are easy to use.

The problem with this approach is that building OBDSSs that are easy to use, may be very costly and time-consuming. The main objective of the GESCOPP project was to design and construct an OBDSS generator that could significantly reduce both the time and the cost of constructing easy-to-use OBDSSs. This generator uses SML to represent the optimization model of the OBDSS. Given the model in SML, the generator can produce an application within a few hours, with very little human intervention. By the end of the GESCOPP project, in 1998, we had a fully functional version of this generator. Since then, the generator has been used to build a number of OBDSSs, which are currently being used by several Chilean firms.

Our overall assessment, after this experience, is that the generator can greatly reduce both the cost and the development time of constructing OBDSSs that are powerful, easy to use, and easy to maintain. Three aspects of the design of the generated OBDSSs were, in our opinion, important in achieving this objective: the strict separation between the model structure, the data, and the solver; the use of RDBMSs to handle the detailed data of the model instances; and the use of the instance generator.

The strict separation between the model structure, the data, and the solver allowed us to reuse several key components of the OBDSS; it also allowed us to make use of the best tools for handling the data and generating the graphical user interface. We could achieve this strict separation thanks to the structured modeling approach and the features of the SML modeling language.

The use of a RDBMS for handling the detailed data of the model instance, provides the kind of support that is required to successfully cope with real-world problems. The RDBMS can greatly
help the user to maintain data consistency and integrity of the input data required to solve the problem, and also to view and analyze the results.

The use of the instance generator was probably one of the most important aspects in order to reduce the time required to build an OBDSS. The time required to generate an instance, however, can become too large for big data sets.

Based on our experience with the generator developed during the GESCOPP project, we think that its use is appropriate for developing functional prototypes of OBDSSs, which can be modified later, with relatively little effort, to make them fully functional systems. The architecture of the generated systems makes it possible to replace some of the components without affecting the rest of the system. In the case of many small to medium-sized problems, the generated OBDSS can be used without further modification. This can make the use of this type of system feasible for a large number of potential users, by substantially reducing the development time and cost. We hope this will induce many decision-makers to replace the rudimentary tools they currently use with tools based on OR/MS technology.

Acknowledgments

Support was provided by FONDEF, FONDECYT, and two participating firms, CTI and Dos en Uno. We also appreciate the collaboration of Luis Contesse, Carlos Donoso, Yadran Eterovic, Pedro Gazmuri, Allan Gubbins, and Cristián Villena who worked with us on the GESCOPP project. We also thank the three anonymous referees for their valuable suggestions.
References


