# THE DESIGN AND IMPLEMENTATION OF A PROTOTYPE STRUCTURED MODELING ENVIRONMENT

L. NEUSTADTER, A. GEOFFRION, S. MATURANA, Y. TSAI and F. VICUÑA
*John E. Anderson Graduate School of Management, University of California, Los Angeles, CA 90024, USA*

## Abstract

This paper describes the design and implementation of the prototype structured modeling environment FW/SM. The underlying design principles provide the central focus. Other points of interest include discussions of FW/SM's delivery platform, its interface to external packages, and its optimization interfaces. The intended audience is all modeling system evaluators, designers, and implementors, including those who do not happen to take a structured modeling approach.

## 1. Introduction

Modeling environments have been proposed as part of a solution to the oft-cited productivity, quality, and acceptance problems facing the Management Science/ Operations Research community [13]. A modeling environment is an integrated set of computer-based tools providing comprehensive support for applied model-based work.

This paper describes the design and implementation of a prototype modeling environment called FW/SM developed at UCLA over a period of about five years. FW/SM consists of a set of user-invokable *processes*, each of which provides one functionality or coherent group of functionalities. The underlying design philosophy is that a *modeling environment should be based on an executable language for representing models, which in turn should be based on an explicit conceptual framework for thinking about models*. FW/SM adopts Structured Modeling Language (SML) as its executable modeling language. SML, in turn, is based on the structured modeling (SM) conceptual framework; see Geoffrion [10,12,14].

In the course of this paper, we will describe how we have exploited certain properties of SM and SML in the design of FW/SM. This will permit readers to judge the merits of SM and SML in the context of modeling environment design, as well as provide concrete ideas for the design of other SM-based environments. Exploiting the underlying language and conceptual framework is a central challenge of modeling environment design.

Many of our ideas are transferable to contexts other than SM. For example, FW/SM's design exploits SML's sharp separation of general model structure from instantiating data, a property shared by some other languages. Hence, this paper should be of interest to designers of modeling languages, systems, and environments in general, even if they do not take a structured modeling approach.

We summarize FW/SM's design in terms of four *design principles*. Some of these principles answer the challenge of exploiting opportunities in the underlying language and conceptual framework, whereas others are independent of the foundations. These principles are not original. Indeed, software developers have been using them or variations of them for years. However, few modeling systems appear to abide by them. This paper illuminates their applicability to language-based modeling environments. In brief, these (not necessarily orthogonal) design principles are:

DP1. *Divide processing according to the relative stability of model components.* In SML, as in other languages, some model components are more stable–that is, change less frequently–than others. For example, an SML schema (i.e. the general model structure) tends to be more stable than the associated elemental detail (i.e. the instantiating data). Redundant processing can be avoided by processing relatively stable components once and storing results in an intermediate form for later reuse. In FW/SM, for example, a separately invokable process makes correctness checks on the schema and stores a parsed representation of it for later reuse. These correctness checks and certain types of parsing do not have to be repeated until the schema changes.

DP2. *Do not force users to pay the price of performing multiple tasks when they only need to perform one.* FS/SM makes correctness checks on general model structure, on semi-formal documentation, and on instantiating data through separately invokable processes, thus reducing redundant processing and providing flexibility in how models are developed and used. In contrast, some modeling systems check general structure and instantiating data correctness each time any process is run, regardless of whether any changes have been made. Moreover, in many systems a complete model instance must be present before any processing can be done on it.

DP3. *Provide "core services" as a basis for extensibility and quality.* In order to facilitate modeling environment extensibility and quality, developers wishing to add functionality to the modeling environment should have access to a well designed set of core services. This can be achieved in a variety of ways, including subroutine libraries, abstract data types, and/or access to low-level intermediate results produced by existing processes. FW/SM makes a start toward providing core services via the last approach.

DP4. *Hide details about the user interface from modeling environment processes.* For evolutionary flexibility and other reasons, it is important that it be relatively easy to change aspects of the user interface or even the whole user

interface without modifying any of the underlying code that implements essential modeling environment functionality. FW/SM achieves this goal through carefully specified ASCII file interfaces at the operating system level.

These principles are themes that run throughout this paper. We will describe how FW/SM realizes them and argue their benefits.

Other points of interest in this paper include discussions of FW/SM's delivery platform, its interfaces to external, interactive systems, and its optimization interfaces.

This paper does not describe the functionality of FW/SM; that is done in the companion paper [15] and in Geoffrion et al. [17]. Nor is this paper intended to serve as technical documentation for FW/SM; that is done in a series of internal documents (see Geoffrion et al. [16]). Rather, our purpose is to highlight the aspects of FW/SM's design and implementation which seem most likely to be of general interest and that can be adapted to other situations.

Full comprehension requires the reader to have prior exposure to SML at the level of sections 1–3 of Geoffrion [10], and to FW/SM as presented in the companion paper [15]. The appendix gives a brief description of all pertinent FW/SM processes for reference purposes. It is assumed that readers have rudimentary knowledge of computer science. Familiarity with [13] and [14] would be helpful but is not necessary.

During the course of this paper, we will occasionally contrast FW/SM's design with that of other modeling systems. Our purpose is to clarify FW/SM's design, and to show that FW/SM differs from at least some other systems. We focus on the mathematical programming systems AMPL [9] and GAMS [4] because they are relatively well known and well documented, and because they have enough in common with FW/SM to invite meaningful comparisons. Our comparisons should not be construed as criticisms of these other systems, which are excellent in their own right.

The organization of the balance of this paper is as follows. The remainder of this introduction discusses the terms *modeling environment* and *software integration*. Section 2 discusses how FW/SM has been influenced by the foundations of SM and the nature of SML. Section 3 discusses FW/SM's overall architecture. Section 5 provides details on selected FW/SM processes. Finally, section 6 concludes by summarizing how FW/SM realizes the aforementioned design principles, and by offering some final reflections based on our experiences with FW/SM.

### 1.1. MODELING *ENVIRONMENTS*

In our view, a modeling *environment* is marked by three distinguishing characteristics. First a modeling environment can accommodate multiple modeling paradigms – not just inventory, or Markov, or network, or project management, or queueing models. Second, it offers multiple solver types – not just expression evaluators, or inference engines, or optimizers, or query engines, or simulators. Third, it aims

to support the entire modeling life-cycle – not just one or two particular phases (such as the "solution" phase, which may account for only 10% of the life-cycle). See Geoffrion [13] for an elaboration of these ideas.

FW/SM approximates a modeling environment in the following ways. First, FW/SM accommodates multiple modeling paradigms, since it is based on the paradigm-neutral language SML. Second, diverse solvers have been integrated into FW/SM – namely, an expression evaluator, a generalized network flow optimizer, a linear/integer program optimizer, a Prolog query engine, and a relational database query engine. Third, FW/SM offers a variety of functionalities that support different phases of the modeling life-cycle. For example, FW/SM's automatically generated reference documentation supports evolutionary change, and its automatic link to a relational database system supports pre- and post-solver inspection of instantiating data and results. See the appendix for a summary of FW/SM's functionalities. In addition, FW/SM offers an integrated set of standard utilities (e.g. graphing, printing, spreadsheets, telecommunications, and word processing) that are useful throughout the modeling life-cycle.

### 1.2. SOFTWARE INTEGRATION

Intergrating diverse functionalities into a single software system is one of the major challenges of modeling environment design. Software integration issues can be divided into *external* and *internal* ones [27]. External issues pertain to the user interface, whereas internal ones pertain to the underlying code that implements essential environment functionality.

Successful external integration means that users face a conceptually consistent interface. For example, the same editor should apply in all parts of the environment. External integration impacts modeling environment usability.

Successful internal integration means that processes share information whenever possible. For example, a process should not bother the user for a piece of information that has already been input to the environment. Nor should it needlessly recompute it. Also, processes should share underlying components whenever possible. Internal integration impacts modeling environment performance, size, quality, and extensibility.

FW/SM approaches software integration in two orthogonal ways. External integration is achieved by choosing a paradigm-neutral modeling framework and by building on top of a highly integrated multi-function software package, namely Framework IV [3]. Internal integration is achieved by careful architectural design. We elaborate on our approaches to external and internal integration in sections 3 and 4, respectively.

## 2. Influence of the underlying foundations

At the beginning of section 1, we stated our belief that a modeling environment should be based on an executable language for representing models, which in turn

should be based on an explicit conceptual framework for thinking about models. This section explains how FW/SM's design reflects this approach, although many details must of necessity be postponed to later sections. Although this section focuses mainly on SML, we hope that readers will think about their own favorite modeling language and how it either facilitates or hinders specific capabilities in systems that embody it.

We first discuss the influence of SML's underlying conceptual framework on FW/SM's design, and then explain how SML influenced its design. In addition, SML strongly influenced our choice of an implementation platform, as discussed in section 3.

FW/SM is unusual, and perhaps even unique, in that it is based on a fully explicit conceptual framework [12]. The underlying SM framework gives a central role to capturing definitional dependencies when specifying the general structure of a model. This makes a number of FW/SM capabilities possible, including:

- exhaustive error checking with respect to the underlying SM framework,
- a universally applicable graphical style of model representation,
- certain types of reference documentation, and
- SML's interpretation sublanguage.

These capabilities are embodied in the FW/SM processes *SMLCheck*, *NETDRAW*, *REFGEN*, and *CK_INTERP*, respectively, all of which depend heavily on the existence of definitional dependencies. See Geoffrion [14] for more discussion on the significance of definitional dependencies, and see the appendix for brief descriptions of these processes.

We now describe the structure of a typical SML model and explain how FW/SM exploits this structure.

### 2.1. SML MODEL COMPONENTS

Recall (e.g. from Geoffrion [10]) that an SML model consists of a *schema* and a set of *elemental detail tables*. The schema divides into a *formal part* and an *interpretation part*. The formal part describes the general structure of a class of model instances, and the interpretation part documents the formal part. The elemental detail tables provide the instantiating data for a single model instance within the model class described by the associated schema. The elemental detail tables can be viewed as consisting of a (relational) database scheme and a (relational) database instance. In FW/SM, the database schema is called the *table structure*, and the database instance is called the *detailed data*.

Figure 1 shows the dependency relationships among these SML model components. The interpretation part of the schema is constrained by the formal part and certain composition rules. The table structure is constrained by the formal part of the schema and so-called *table structuring procedure* given in Geoffrion [11].

The detailed data must be organized according to the table structure and must be consistent with all specifications given by the schema (the schema specifies constraints on the data types, index set populations, etc.).

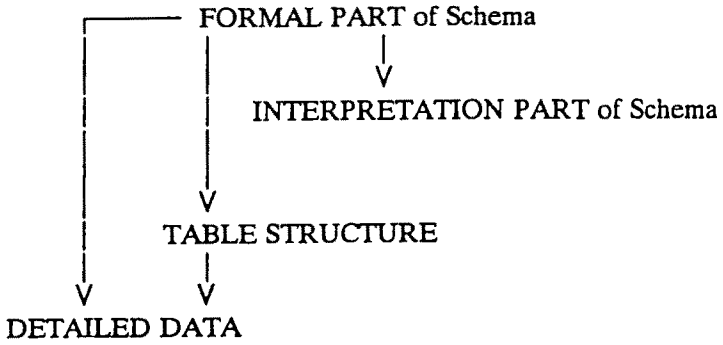


Fig. 1. SML model components
(A → B means that A constrains B)

## 2.2. INFLUENCE OF SML ON FUNCTIONAL DESIGN

FW/SM's functional design reflects the structure of SML models. There are FW/SM processes that target the formal part of the schema only, the interpretation part only, the table structure only, and the detailed data only. Of course, there are also FW/SM processes which operate on combinations of these. We first map selected FW/SM processes to the SML model components, to give the reader a feeling for FW/SM's functional design. We then comment on the significance of the functional design.

A number of FW/SM processes check and analyze the formal part of the schema:

- *SMLCheck* checks the syntax and semantics of the formal part of the schema.
- *REFGEN* generates documentation for the formal part of the schema, and *DBREF* transforms some of this documentation to an alternative database-accessible format.
- *NETDRAW* produces a graph of the definitional dependencies given in the formal part of the schema.
- *PrologQuery* enables answering ad hoc logic-based queries about certain aspects of the formal part of the schema.

*CK_INTERP* checks the interpretation part of the schema. *SMLCheck* and *CK_INTERP* are separately invokable processes, so that modelers can iteratively refine interpretations after the formal part of the schema has stabilized, without paying the price of re-checking the formal part. (Since the interpretation part of the schema depends on

the formal part, *SMLCheck* must have run successfully before *CK_INTERP* may be invoked.)

*EDGEN* creates the table structure and the elemental detail table skeletons in accordance with SML's table structuring procedure. (Since the table structure depends on the formal part of the schema, *SMLCheck* must have run successfully before *EDGEN* may be invoked.)

Detailed data checks, which have not yet been fully implemented, are invokable separately from *SMLCheck* and *CK_INTERP*. Hence, modelers need not pay the price of re-checking the formal or interpretation part of the schema when only the detailed data have changed. This has the potential to greatly reduce processing time, since the detailed data tends to be much more volatile than the schema.

Finally, many processes require multiple SML model components. For example, most solvers (e.g. *MPS_INTERFACE* and *XtrieveQuery*) require the schema, table structure, and the detailed data. Lexical, syntactic, and semantic error checking is not incorporated into these processes; it is assumed that the model has passed these types of checks.

There are two important points to note about the preceding discussion:

(1) FW/SM gives modelers the flexibility to develop models in stages, checking correctness after each stage is completed. This exposes errors earlier than would be the case if an entire model instance had to be developed before doing any error checking or analysis. In contrast, some modeling systems (e.g. GAMS) require both model structure and instantiating data to be present before any processing (including error-checking) can be done.

(2) Correctness checks on the formal part of the schema, the interpretation part, and the detailed data are all separately invokable processes. Furthermore, these kinds of checks are not incorporated into solver interfaces, as they are in traditional modeling systems. This eliminates much redundant processing. In contrast, some modeling systems (e.g. GAMS) check general structure and instantiating data correctness each time any process is run, regardless of whether any changes have been made.

Clearly, the structure of SML models shown in fig. 1 facilitates this kind of functional design. For example, many aspects of this design would be difficult if not impossible to achieve in a modeling system aimed at supporting a language that does not make a sharp distinction between general model structure and instantiating data. Note that FW/SM's functional design is consistent with the first two design principles ("divide processing to the relative stability of model components" and "do not force users to pay the price of performing multiple tasks when they only need to perform one").

### 2.3. INFLUENCE OF SML ON SOFTWARE INTERNALS

Thus far, our focus has been on the functional design of FW/MS. SML's structure also influenced the design of FW/SM's software internals. This influence

has been in two areas to which we now turn: schema parsing and expression evaluation.

In addition to checking the formal part of the schema, *SMLCheck* creates an internal tabular representation of the parsed schema for use by all other processes. This internal representation is stored in external ASCII files for later access, saving other FW/SM processes the work of doing this parsing. Since the parsed representation is essentially a rearrangement of *SMLCheck*'s internal symbol table, the extra processing time required is negligible. This leads to savings in development time (developers do not have to implement this parser), saves processing time (this parsing is done once per schema change), and improves quality (*SMLCheck* has been thoroughly tested). Note that the savings in processing time are dependent on the relative stability of the schema, which is a direct consequence of separating general structure and instantiating data.

In FW/SM, the task of evaluation is split into two processes: *FcEval* and *Evaluate*. The former compiles the generic rules of function and test genera into executable C code and depends only on the schema and table structure (the organization of the data is hard-coded into the compiled program). The executable code is saved in an external file and need only be produced once per schema or table structure change. *Evaluate* does the actual evaluation by invoking the executable code produced by *FcEval*, which operates on the detailed data to produce the evaluation results. Again, this approach exploits the relative stability of the schema and table structure.

Note that both of the preceding examples are consistent with the first design principle.

There is one potential problem with FW/SM's design. The modeler is responsible for insuring that SML models are correct (i.e. have passed *SMLCheck*, *CK_INTERP*, and all detailed data checks). In particular, the modeler must remember to re-invoke the appropriate check whenever any changes are made to the schema or detailed data. Similarly, the modeler must remember to re-invoke *FcEval* whenever any changes are made to the schema or table structure. Software engineers have long recognized and remedied problems like this, and the configuration management technology that has been developed for this purpose can be adapted to modeling environments. For example, a "make"-like facility (e.g. Feldman [8]) could be incorporated into FW/SM to automatically re-run all neccessary processes whenever a model component has changed.

### 2.4. BENEFITS OF TABLE STRUCTURE

We make some remarks concerning the importance of table structure, which is an explicit database scheme mapping the SML schema to the instantiating data. (Note that SML goes further than simply separating general model structure from instantiating data.) This is highly unusual among model definition languages. In some languages (e.g. AMPL and GAMS), the database scheme is only implicit in the data section of the model. Having an explicit database scheme makes data-

driven processes more efficient, since processes can use it to quickly locate data. In contrast, the input data to AMPL and GAMS must be compiled or interpreted before it can be accessed and manipulated. In addition, an explicit database scheme opens the door to processes that manipulate data only, such as *XtrieveQuery*.

### 2.5. BENEFITS OF SML'S COMPREHENSIVE SPECIFICATION

Finally, we offer some comments on the importance of a language specification. Although some of these comments are not pertinent to FW/SM per se, we offer them because they are highly pertinent to modeling environment implementation in general.

SML is unusual among model definition languages in that it is comprehensively specified. Geoffrion [11] completely specifies syntax and semantics for both general model structure and model instance data. Henceforth, this specification will be referred to as the *Working Paper 360 (WP 360) specification.* Although the importance of a rigorous and complete language specification has long been recognized in the programming language community, it has not yet been widely recognized in the modeling community. Such a specification offers many benefits.

First, a comprehensive specification saves implementation time. WP 360 specifies exactly how SML implementations should behave with respect to commonly unspecified semantics, such as how to deal with missing objects, missing values, degenerate or unanticipated cases, and other details. We wasted no time pondering semantics in those instances.

Second, independent implementations of SML meeting the WP 360 specification will all have the same specified semantic behavior. For example, independently implemented SML compilers should accept and reject the same inputs. (In particular, all models accepted by SML compilers meeting the WP 360 specification are guaranteed to be "genuine" structured models, by propositions 1 and 2 in Geoffrion [11].) For another example, all solvers incorporating evaluation in SML-based environments will exhibit the same specified run-time semantics (e.g. treat missing elements in the same way). This is particularly important in modeling environments, where multiple solvers are supported.

Third, a modeling language with a comprehensive specification tends to be less susceptible to undetected modeling errors arising from improper usage. The less comprehensive a language's specification, the more incorrect assumptions modelers are likely to make about the behavior of the language because of the greater reliance they must place on empirical testing as a way to resolve questionable aspects of language behavior.

Finally, a comprehensive specification facilitates using automated tools to generate parts of modeling environments. For example, Vicuña [30] used an attribute grammar to specify formally the complete static semantics of SML, and therefrom to generate a syntax-directed editor with incremental type checking and immediate expression evaluation capabilities. This editor was built using the *Synthesizer Generator* [25].

## 3. FW/SM delivery platform

FW/SM delivers modeling environment functionality through the integrated multi-function program Framework IV. Geoffrion [15] describes Framework, our reasons for choosing it, and FW/SM's user interface. This section highlights the role Framework plays in FW/SM's architecture. We point out how FW/SM's Framework-based user interface exploits similarities between SML and Framework, and provides a high level of external integration. We then summarize the advantages of building on top of a package such as Framework.

### 3.1. EXPLOITING SIMILARITIES BETWEEN SML AND FRAMEWORK

By building on top of Framework, we were able to achieve easily a user interface that mirrors and supports important aspects of the underlying language and conceptual framework:

- FW/SM represents the schema as a Framework outline that corresponds exactly to the model's modular outline, a structure easily created, edited, and browsed using Framework's tree-oriented editor. In addition, this editor can be used to selectively hide and unhide schema subtrees, a feature that provides support for hierarchical model organization.
- FW/SM represents each elemental detail table as a Framework database frame. Framework's table-oriented editor makes it easy to enter, modify, and browse data, query single tables, and change the format in which the data are displayed.

### 3.2. EXTERNAL INTEGRATION

Framework is responsible for FW/SM's high degree of external integration. Not only does Framework provide a set of data abstractions capable of naturally representing SML models and associated objects, but it also provides a comprehensive set of utilities that can be applied to them in consistent ways. For example, word processing utilities are invoked and behave in exactly the same way regardless of the type of object to which they are applied. Since all FW/SM modeling objects are represented as Framework data abstractions, and all FW/SM processes read from and write to Framework, the FW/SM user enjoys a highly consistent user interface.

### 3.3. COMPARISONS WITH OTHER SYSTEMS

We have explained how model structure, instantiating data, and related objects are represented quite naturally by Framework data abstractions which are accessible by common utilities in consistent ways. In other words, the FW/SM user interface exploits special structure, yet is appropriately consistent.

In order to see the significance of this, it is useful to make comparisons with other modeling systems. Many systems achieve consistency by representing all

modeling objects in a single way. For example, many mathematical programming systems based on an algebraic language (e.g. AMPL and GAMS) represent all objects as text. A word processor offers consistent access to these objects, but fails to exploit their special structure. Another example is provided by systems which represent models in their entirety as data in a database management system (e.g. Lenard [19]). The host data manipulation language then applies uniformly to both model structure and instantiating data. Although this second approach provides powerful manipulation capabilities, it has the significant disadvantage of forcing users to think about model structure in terms of a *data model*, usually the relational one, which is somewhat unnatural. The FW/SM approach seems to achieve a better balance between naturalness of representation, consistency of access, and exploitation of special structure.

### 3.4. ADVANTAGES AND DISADVANTAGES OF BUILDING ON FRAMEWORK

The advantages of building on Framework have been explained in the companion paper [15] and in the preceding discussion. Briefly, Framework saved a tremendous amount of development work. Among other things, it enabled us to achieve FW/SM's user interface and standard utilities at little development cost; we merely had to recognize the similarities between Framework's data abstractions and SML, and then to design the FW/SM user interface so as to exploit those similarities.

These advantages come at a price. Framework's most serious shortcoming is its weak support for databases, which is inherited by FW/SM. In addition, Framework's largely inaccessible internals limit some types of extensibility. Finally Framework is avilable only on DOS-based systems, thus restricting FW/SM's portability. We offer some further comments in this vein at the very end of this paper.

## 4. FW/SM architecture

This section describes FW/SM's overall architecture: section 4.1 gives a high level overview, section 4.2 discusses the architecture of FW/SM's core operating environment, and section 4.3 discusses FW/SM's interfaces to external systems.

### 4.1. OVERVIEW

FW/SM runs on IBM-compatible personal computers under DOS. Framework and DOS form the core operating environment of FW/SM. Framework is the dominant FW/SM user environment; it is "home" to the FW/SM user. In addition, the FW/SM architecture accommodates optional automatic interfaces to stand-alone, external systems. There are two such interfaces. One, *XtrieveQuery*, is to the relational database program Xtrieve [23], and the other, *PrologQuery*, is to the Arity Prolog language and inference engine [2].

Figure 2 gives a high level overview of FW/SM's architecture. From an implementation perspective, there are four distinct software environments within FW/SM: Framework, DOS, Arity Prolog and Xtrieve. FW/SM programs running in these four environments implement the special functionalities of FW/SM. We first explain how control is passed from one environment to another, and then how information is passed.

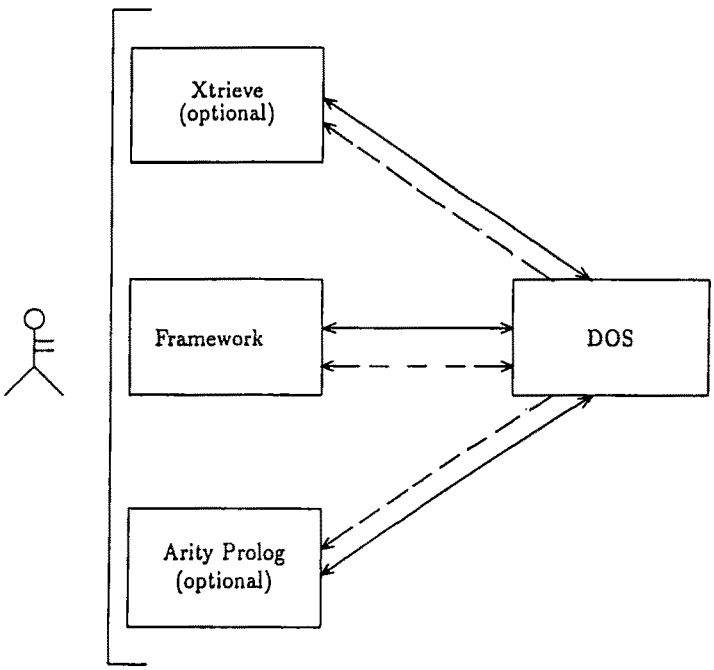


Fig. 2. FW/SM software environments.

### 4.1.1. *Flow of control*

The user accesses FW/SM via a macro from within Framework by pressing a special key. This invokes the FW/SM *Shell*, which displays a non-obtrusive, two-line menu at the bottom of the screen listing the FW/SM processes from which the user may select. All FW/SM processes fall into one of four processing categories:

(1) all processing is done in Framework;

(2) processing is split between Framework and DOS;

(3) processing is split between Framework, DOS, and Xtrieve (*XtrieveQuery* only);

(4) processing is split between Framework, DOS, and Arity Prolog (*PrologQuery* only).

Most FW/SM processes fall in the second category. FW/SM programs running in Framework pass control to the DOS environment by executing a DOS batch file (a script of DOS commands). Upon completion of the batch file, control returns to the Framework environment.

In the case of *XtrieveQuery*, control passes from Framework to DOS and then to Xtrieve. Once in Xtrieve, control passes to the user and the full power of Xtrieve becomes available. When the user exits Xtrieve, control passes automatically through DOS back to Framework. *PrologQuery* processing is analogous.

#### *4.1.1. Flow of information*

FW/SM programs running under DOS require information from the Framework environment. All required information is written to ASCII disk files via Framework's native export capabilities. After control passes to DOS, the information is available in a format that can be read by FW/SM programs running under DOS. Information flow in the other direction – from DOS to Framework – is analogous. FW/SM programs running under DOS write user output to ASCII disk files. After control passes to Framework, FW/SM programs import these files using Framework's native import capabilities. FW/SM programs running within Framework then copy the imported data to the appropriate user workspace.

The communication mechanism between Framework and Xtrieve is via DOS. In DOS, exported Framework data (e.g. elemental detail tables) are transformed to a format readable by Xtrieve. Xtrieve is invoked from DOS; upon entry to Xtrieve, a FW/SM program sets up the Xtrieve environment for the current model. Currently there is no information flow from Xtrieve to DOS, and hence none from Xtrieve to Framework. Consequently, any changes the user makes to model data within the Xtrieve environment are not reflected in the Framework environment. Communication between Framework and Arity Prolog (via DOS) is similar.

### 4.2. CORE OPERATING ENVIRONMENT

This subsection discusses FW/SM's core operating environment: salient aspects of its architecture, the schema's internal representation, and the mechanism by which model-specific intermediate results are stored and later reused.

#### *4.2.1. Introduction*

There are two important points to understand about the architecture. The first concerns the input/output relationships among FW/SM processes, and the second concerns the standard interface between the Framework and DOS processing environments.

(a) *Process interdependencies.* FW/SM consists of loosely coupled processes communicating via external files. FW/SM processes are interdependent in the sense

that outputs from one process may be inputs to another. The most common type of interdependency is between *SMLCheck* and other FW/SM processes, which we now discuss.

Figure 3 shows *SMLCheck*'s central architectural role. The FW/SM user sees and works with the Framework representation of the schema. When the user invokes *SMLCheck*, FW/SM's *WriteSchema* utility exports the Framework representation of

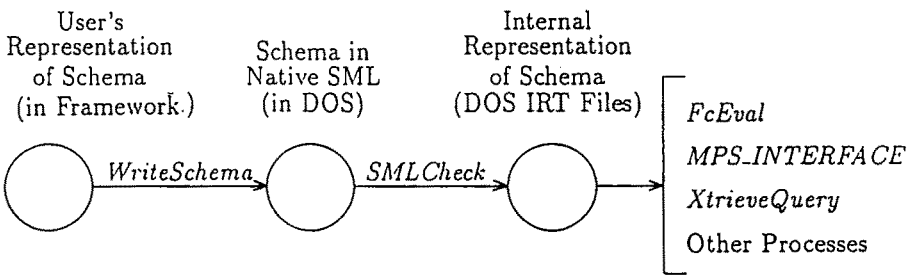


Fig. 3. The role of *SMLCheck* in the FW/SM architecture.

the schema to an ASCII file in the DOS environment. The exported schema is written in *native* SML (i.e. SML per the WP 360 specification of Geoffrion [11]). *SMLCheck* operates on the schema in native SML. It checks for syntax and semantic errors and, if none exist, produces an internal tabular ASCII representation of the schema called the *internal representation tables*. *SMLCheck* can be viewed as the front end of a compiler; the internal representation is essentially a rearrangement of *SMLCheck*'s internal symbol table (see Aho et al. [1] for an overview of compiler theory). All FW/SM processes requiring schema information access the internal representation tables, rather than the schema itself. That is, *SMLCheck*'s outputs are inputs to other processes.

Section 4.2.2 provides more details on the internal representation tables, and section 4.2.3 explains the mechanics by which a process's outputs are stored and subsequently retrieved to be used as inputs to another process.

There are other input/output relationships among FW/SM processes, the most significant of which is the relationship between *FcEval* and *Evaluate*. That will be discussed in section 5.2.2.

(b) *The Framework–DOS interface.* FW/SM programs running under DOS operate on standard internal ASCII representations of modeling objects (as opposed to Framework representations). This helps realize design principle 4. We have already explained how the utility *WriteSchema* generates an ASCII version of the schema written in native SML, upon which *SMLCheck* operates. In addition to the schema, other modeling objects such as detailed data (stored in the elemental detail tables) must be exported from the Framework environment to DOS. Standards exist for how these other types of modeling data are represented under DOS.

All details about the user interface are encapsulated in the utility programs that convert the Framework representation of an SML model into the standard internal representation required by FW/SM programs running under DOS. This permits considerable leeway in the user interface. For example, FW/SM uses fonts to improve SML's readability. Figure 4 shows a schema as it appears in Framework and in native SML. FW/SM processes running under DOS know nothing about the particulars of the FW/SM user interface. The Framework environment could be replaced by *any* user interface; no changes would have to be made to most of the processes, so long as the proper ASCII files are produced. Less radically, changes could easily be made to the FW/SM user interface. Note that this design facilitates user interface customization.

**EVENTe /pe/** There is an ordered list of EVENTS.

**PERM (EVENT<1:e>) /f/ ; e * @IF (e=1, 1, PERM<e-1>)** The number of PERMUTATIONS of the first n EVENTS is n!.

**(a) Schema In Framework**

```
EVENTe /pe/   ¯¦ There is an ordered list of ¯/EVENTS¯/. ¯.

PERM (EVENT<1:e>) /f/ ; e * @IF (e=1, 1, PERM<e-1>)   ¯¦ The number
of ¯/PERMUTATIONS¯/ of the first n EVENTS is n!. ¯.¯.
```

**(b) Schema In Native SML**

Fig. 4. Schema as it appears in Framework and in native SML.

Since most key processes are implemented in C, they can be ported easily to non-DOS systems and linked to the host user interface. (Of course, "linking" requires writing utilities that generate the standard internal representations of required modeling objects – e.g. writing an analog of *WriteSchema*.) To underscore this point, we hereby offer stand-alone versions of most key processes (viz. *FcEval/ Evaluate*, *INTER_CK*, *GENNETFW*, *MPS_INTERFACE*, and *SMLCheck*) to researchers with an active interest in implementing SML-based environments. A few other important processes, mainly *EDGEN* and *REFGEN*, are implemented using substantial amounts of FRED, Framework's internal programming language, and so are not easily portable.

In summary, our manifestation of design principle 4 facilitates customization of the user interface, and offers both portability of core processes and delivery of modeling environment functionality through a (not so portable) popular software package.

### 4.2.2. *Internal representation tables*

The internal representation tables (IRT) are a tabular, parsed representation of the schema that reside in disk in ASCII files (Vicuña et al. [31]). There are 30 tables in all, most of which represent just one aspect of the schema. These tables are complete in the sense that the SML schema can be recreated from them.

The design of the IRT files was largely dictated by the (quasi-)structured model that formally specifies SML's context-free grammar (appendix 6 in Geoffrion [11]). This design corresponds to the elemental detail tables associated with that model's schema. Designing the IRT files in this way tested the utility of SML's table structuring procedure and avoided what would otherwise have been a fairly arbitrary design process.

To give the reader a feeling for what the IRT files are like, fig. 5 shows the contents of one of them for the Feedmix model from Geoffrion [10] or [15]. For each genus and module in the schema, this file contains its position in the modular outline, its name, its type ("m" stands for module), and the genera it calls. The first line in the file is the heading, and the remaining lines are data. For example, the two lines marked by arrows indicate that the seventh entry in the modular outline is a genus named "ANALYSIS" of type "a" (for attribute), and that "ANALYSIS" calls "NUTR" and "MATERIAL".

```
      "G_M_POS","NAME","||","TYPE","CALLED_GNAMES"
      "1","&NUT_DATA","||","m",""
      "2","NUTR","||","pe",""
      "3","MIN","||","a","NUTR"
      "4","&MATERIALS","||","m",""
      "5","MATERIAL","||","pe",""
      "6","UCOST","||","a","MATERIAL"
--->  "7","ANALYSIS","||","a","NUTR"
--->  "7","ANALYSIS","||","a","MATERIAL"
      "8","Q","||","va","MATERIAL"
      "9","NLEVEL","||","f","ANALYSIS"
      "9","NLEVEL","||","f","Q"
      "10","T:NLEVEL","||","t","NLEVEL"
      "10","T:NLEVEL","||","t","MIN"
      "11","TOTCOST","||","f","UCOST"
      "11","TOTCOST","||","f","Q"
```

Fig. 5. Sample IRT file: SYNTHSIS.IRT.

Several alternative internal representation table designs have been proposed by other researchers for a variety of purposes (e.g. Dolk [6], Lenard [19], Tsai [28], and Tung [29]). For example, Tsai's design facilitates schema queries. These researchers propose storing the tables in a relational database management system rather than in ASCII files, an idea that becomes increasingly attractive as such systems become increasingly available.

Whatever the relative merits or demerits of our particular design and storage mechanism, the IRT files have worked out very well in our implementation. They reduced the effort required to add new processes, since developers did not need to implement certain types of schema parsing routines or incorporate syntax or semantic error checks into their processes. The IRT files are guaranteed to represent a correct schema.

A truly extensible modeling environment needs to provide "core services" for new developers, as advocated by design principle 3. The IRT files are an example of this idea, although we now recognize that we should have built additional core services into FW/SM. For example, a library of routines to access the IRT files would have been valuable (currently, each FW/SM process provides its own access routines).

#### *4.2.3. Auxiliary files*

*Auxiliary files* are model-specific intermediate results that are stored on disk for later use. Intermediate results that depend only on the schema and/or table structure are good candidates to become auxiliary files, due to the relative stability of these components (such results do not have to be regenerated until the schema or table structure is modified). The purpose of auxiliary files is to reduce redundant work, thus giving a type of incremental processing capability. Examples are the IRT files, which are used by almost all FW/SM processes, and the executable code produced by *FcEval*, which is used by *Evaluate* (to be discussed in section 5.2.2). Saving intermediate results appears to be unusual among existing modeling systems, many of which take an all-or-nothing approach to processing.

Auxiliary files are stored in model-specific subdirectories, collectively called the *AUXIL* subdirectories, which are created automatically by FW/SM. Processes store and retrieve results to and from these subdirectories. All of this activity is transparent to the user.

### 4.3. INTERFACES TO EXTERNAL SYSTEMS

FW/SM's architecture of loosely coupled processes easily accommodates interfaces to external systems. Currently, FW/SM provides fully automatic interfaces to the database management system Xtrieve and to Arity Prolog.

Such interfaces are important for two reasons. First, they permit achieving modeling environment functionality that would be impractical to implement from scratch. Second, if some users are partial to a specific stand-alone system, then providing an interface to that system may be prudent from an acceptability standpoint. For example, analysts can do modeling studies within the modeling environment itself, and then export the results to an external, locally popular system for perusal by the client (e.g. a manager).

An important issue associated with such interfaces is whether changes made to model data in the external system should be reflected in the home environment.

On the one hand, if this is not done, the model data in the two systems can become inconsistent. On the other hand, it is not clear that it is desirable for modeling data in the home environment to be overwritten automatically. In FW/SM, both *XtrieveQuery* and *PrologQuery* are one-way streets – data are exported to external systems, but never imported. Probably this issue can be resolved only on a case by case basis.

## 5. FW/SM processes

### 5.1. IMPLEMENTATION SUMMARY

Table 1 gives selected implementation information about FW/SM processes. The numeric entries show "lines of source code", where this is the total number of lines in source code files, including blank lines, comments, programming statements split over multiple lines, etc. Some comments are appropriate.

First, note FW/SM's use of embedded packages, an approach that greatly reduced development effort. The purpose of each embedded package is reasonably obvious, except perhaps in the case of *FcEval* and *Evaluate*, companion processes that use a database management system and C compiler (see section 5.2.2).

Second, note that *SMLCheck*, *FcEval*, and *MPS_INTERFACE* all use the lexical analyzer Lex (Lesk [20]) and the parser generator YACC (Johnson [18]). These two tools were very helpful in the implementation of processes that are essentially SML translators.

Third, note that relatively few processes are implemented entirely in FRED, Framework's built-in programming language. For the most part, such processes are tightly coupled with the user interface and must out of necessity be done in FRED. Many processes use FRED only for exporting and importing modeling data, and do the bulk of processing at the operating system level using commercial programs or programs implemented using C or Pascal. Implementing processes in this way promotes modifiability and portability. It also greatly enhances execution speed: FRED is rather slow, since it is interpreted. Moreover, we have found programs written in C to be much easier to enhance and maintain than FRED programs.

In general, the number of lines of code shown for a process is roughly proportional to the effort required to design and implement the process. *SMLCheck*, *FcEval*, and *MPS_INTERFACE* (i.e. the SML translators) are the most complex processes and took the longest to implement.

Finally, we give the reader an idea of the size of FW/SM. The entire system takes 2 MB, excluding Framework and the embedded software packages shown in table 1. The total size of all FW/SM processes, including Framework and the embedded packages, is about 4 MB.

### 5.2. IMPLEMENTATION DETAILS FOR SELECTED PROCESSES

This subsection provides implementation details for selected processes: FW/SM's schema checker, the resident evaluator, the optimization interfaces, and the interfaces to stand-alone, interactive systems.

Table 1

Selected details for FW/SM processes.

| Process | C[a] | FRED | Pascal | YACC | Lex | PROLOG | Embedded Packages |
|---|---|---|---|---|---|---|---|
| *COSMET* | | 160 | | | | | |
| *DBREF* | | 200 | | | | | |
| *EDGEN* | 3,080 | 1,150 | | | | | |
| *Evaluate* | 1,040 | 370 | | | | | dbVista[d] (Raima [29]) |
| *FcEval* | 7,440 | 120 | | 1,880[b] | 570[c] | | dbVista<br>QuickC (Microsoft [22]) |
| *FORMAT* | | 190 | | | | | |
| *GENNETFW* | | 220 | 2,200 | | | | GENNET (Brown and McBride [5]) |
| *ID_DICT* | 90 | 220 | | | | | |
| *INTERP_CK* | 550 | 20 | 1,220 | | | | |
| *L/E Utilities* | | 5,120 | | | | | |
| *MATRIX* | | 280 | | | | | |
| *MPS_INTERFACE* | 17,110 | 530 | | 350[e] | 260[f] | | LINDO (Schrage [26]) |
| *NETDRAW* | 540 | 40 | 2,110 | | | | |
| *PrologQuery* | 500 | 20 | | | | 240 | Arity Prolog (Arity [2]) |
| *REFGEN* | 1,830 | 240 | 360 | | | | |
| *SMLCheck* | 10,990 | 140 | | 1,750[g] | 530[h] | | |
| *TABLE RULES* | | 450 | | | | | |
| *XtrieveQuery* | 1,830 | 50 | | | | | Xtrieve (Novell [23]) |
| System code | 1,860 | 2,360 | | | | | Framework IV |
| Installation Program | 920 | 180 | | | | | |
| TOTALS: | 47,780 | 12,060 | 5,890 | 3,980 | 1,360 | 240 | |

[a] Line counts do not include code generated by LEX and YACC.
[b] Generates 2500 lines of C.
[c] Generates 3110 lines of C.
[d] dbVista is also sold under the name db_FILE.
[e] Generates 980 lines of C.
[f] Generates 1110 lines of C.
[g] Generates 2280 lines of C.
[h] Generates 3900 lines of C.

### 5.2.1. Syntax and semantic checks: SMLCheck

*SMLCheck* checks both the syntactic and semantic validity of an SML schema according to Geoffrion [11], and also breaks it down into IRT files. The semantic checks are written in C using the action routine approach (e.g. Aho et al. [1]) in a parser generated by YACC. The lexical analysis is handled by a scanner module generated by Lex. The parser for SML is large, having 387 grammar rules. Most FW/SM models compile in under a minute on a PS/2 Model 80.

The semantic action approach taken by *SMLCheck* is tailored to the syntax and semantics of SML. All action routines are ad hoc, and so are not easy to maintain and modify. The two major revisions to SML required substantial development effort.

Vicuña [30] investigates an alternative development approach based on attribute grammars. As part of this work, an SML compiler was implemented that was also subject to the SML revisions. The attribute grammar based compiler was much easier to modify than *SMLCheck*, due to the modularity of the specifications of syntax and semantics and the declarativeness which hides all implementation details.

### 5.2.2. Expression evaluation: FcEval and Evaluate

*FcEval* and *Evaluate* are companion processes that together make up FW/SM's expression evaluator for function and test elements. Recall that *FcEval* compiles a schema's generic rules into executable C code, and *Evaluate* executes this code. We go into some detail in this subsection because we believe that expression evaluation capability is important – both for its own sake, and because the ubiquity of spreadsheet programs has created a demand for it.

*FcEval* is our second attempt at designing and implementing an evaluator. Our first attempt was a process coded entirely in FRED that had to be abandoned owing to poor execution performance. Two factors contributed to that failure: (1) the inherent slowness of FRED, and (2) Framework's lack of fast access methods for database frames.

As a result of that experience, we opted for a compiled approach which operates in batch mode. Detailed data are exported to disk, evaluation is done, and the results are imported back into Framework.

*Evaluate* makes use of an internal database manager, the sole purpose of which is to provide efficient access to the detailed data. The physical database is not persistent; it is created and then deleted at each invocation of *Evaluate*. We chose dbVista for this purpose because it (a) can be compiled in C, (b) offers efficient B-tree access, (c) comes with source code that we were able to tailor, (d) offers sequential access and access to first record, last record and to random record through key value(s), and (e) gives the total number of records in a table. All these features are needed during evaluation of SML expressions. Using an internal database manager saved development time and enabled us to concentrate on the application itself rather than on the design of the manipulation and control of the database.

Figure 6 shows *FcEval*'s processing steps. The schema (i.e. the IRT files) and the table structure are used to produce the C source code, which makes calls to the internal database scheme; the source code is compiled and linked. The resulting executable code, named *fceval.exe*, and the internal database scheme are stored in a model-specific AUXIL subdirectory for later use by *Evaluate*. The internal database scheme mirrors the table structure, but is written in dbVista's database definition language and is translated into C by a dbVista utility. The C code calculates the values of function and test elements, drawing values from and putting results back into an internal database instance described by the internal database scheme. Note that *FcEval* does not need to be re-run if only the detailed data have changed.

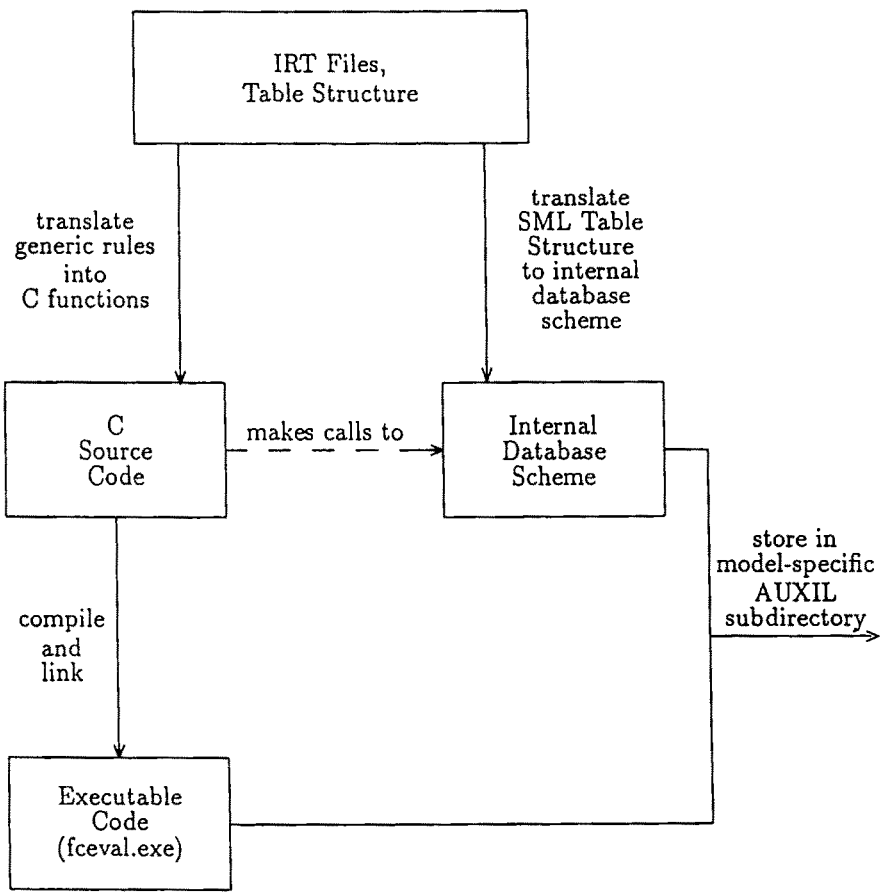


Fig. 6. Data flows for *FcEval* (all processing done in the DOS environment).

Figure 7 shows *Evaluate*'s processing steps. When the user invokes *Evaluate*, the detailed data are exported to disk. *fceval.exe* and the internal database scheme are retrieved from the AUXIL subdirectory. An internal dbVista database holding the detailed data is created and loaded from the exported data and the internal

database scheme. *fceval.exe* then does the evaluation, making calls to the internal database to retrieve data and store the results. After *fcval.exe* completes, the internal database contains all of the results. The results are then imported back from the database into Framework, and all of the function and test columns of the elemental detail tables are updated. *Evaluate* takes an "all-or-nothing" approach; i.e. it has no incremental updating capabilities.

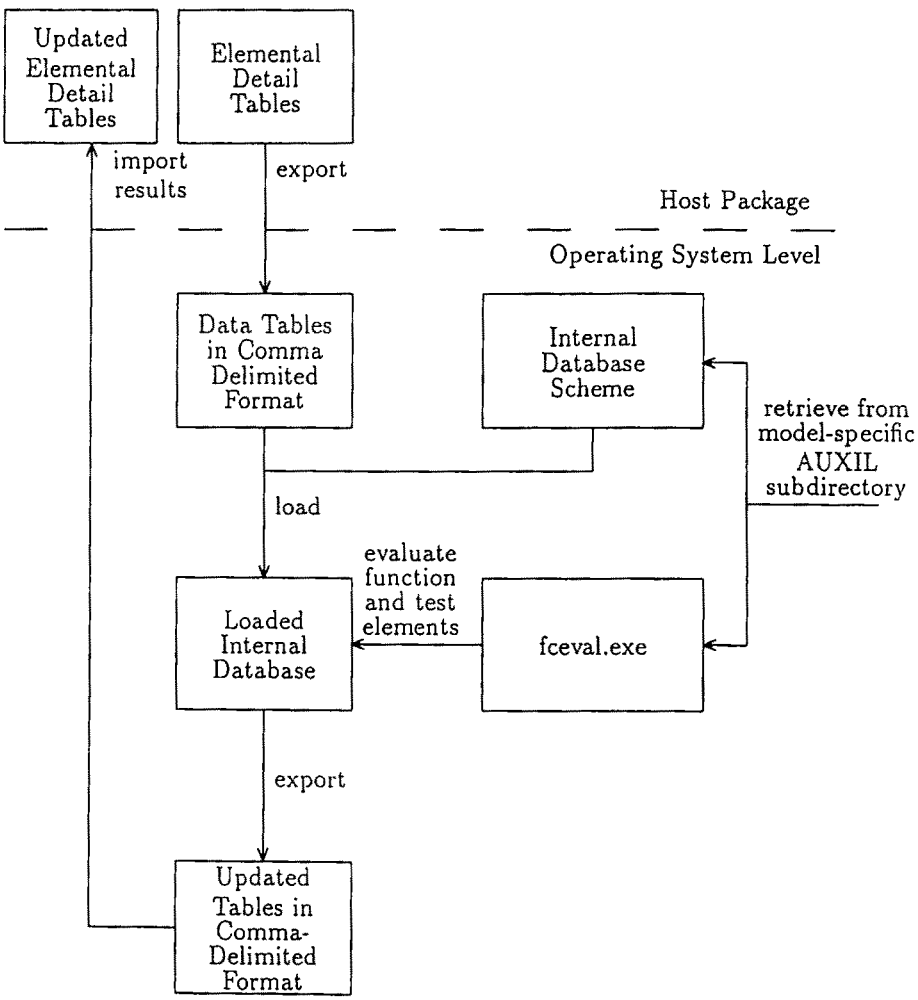


Fig. 7. Data flows for *Evaluate*.

An important point is that *Evaluate* deposits its results directly into the elemental detail tables. This is very convenient for users. Inputs and outputs are stored together and can be manipulated easily using *XtrieveQuery* and Framework's native capabilities.

### 5.2.3. Optimization interfaces: MPS–INTERFACES and GENNETFW

FW/SM has two optimization interfaces, namely *MPS_INTERFACE* and *GENNETFW*. The first is an interface to the mathematical programming system LINDO (Schrage [26]), while the latter is an interface to the generalized flow optimizer GENNET (Brown and McBride [5]). Both solvers run transparently to the user.

Both interfaces communicate with their solvers via external files, a common way to integrate solvers into modeling systems (e.g. Drud [7]). In both cases, an exported SML model instance is translated into a solver input file, the solver is invoked, and the solver output is then translated back into SML-specific form (e.g. internal variable names are translated back into SML names). The results are then imported into Framework, where the user can manipulate them in a variety of ways using Framework's native capabilities.

Within this common framework, *MPS_INTERFACE* and *GENNETFW* represent two different approaches to solver interfaces. We describe and then contrast these approaches. In so doing, we highlight issues pertinent to solver interface design and identify topics for future research.

(a) *MPS_INTERFACE*. This is fully automatic. The modeler expresses an LP or integer LP model in SML, and then *MPS_INTERFACE* translates the SML representation to that required by the solver. *MPS_INTERFACE* assumes that all variable attribute genera are variables and, unless otherwise specified, that all test genera are constraints in the sense that they must evaluate to true. The objective function and direction of optimization are specified in a special control area of the Model Workspace.

*MPS_INTERFACE* has the general architecture of a compiler (e.g. Aho et al. [1]). The schema, table structure, and detailed data are used to construct an MPS format input file. Intermediate expressions are automatically evaluated, and it is verified that the problem is indeed linear. Details about the various logical phases of the translation process can be found in Maturana [21].

*MPS_INTERFACE*'s architecture has much in common with that of mathematical programming systems such as AMPL and GAMS. A major difference is that those systems incorporate lexical, syntactic, and semantic error checks into the solver interface, whereas FW/SM need not, since *SMLCheck* performs these tasks.

(b) *GENNETFW*. This represents a different approach to solver interfaces. *GENNETFW* uses a fixed-structure control table, a declarative device by which the user maps the data in the elemental detail tables into a GENNET input file (Geoffrion [15]).

*GENNETFW* is a low level interface. The person who fills in the control table must know the input data requirements of GENNET, which implies familiarity with GENNET's (simple) input file structure. However, once the control table is set up, the interface is extremely easy to use: one need only populate the elemental detail tables and press a key.

It is important to understand that *GENNETFW* does not refer directly to the schema. The schema serves only as a kind of user documentation for a control table. This is both an advantage and a disadvantage. The advantage is that, since the control table depends only on the schema and table structure, it does not need to be changed as long as these do not change. This is yet another manifestation of the first design principle. The disadvantage is that the control table's independence from the schema deprives it of some of the model context needed to help avoid user error. A grammatically correct control table always represents *some* network problem, but that problem may not be the one intended.

(c) *Selected contrasts.* The *GENNETFW* approach has both advantages and disadvantages relative to the *MPS_INTERFACE* approach. We now contrast these two approaches in terms of *ease of implementation* and *processing speed.*

It is *much* easier to implement a *GENNETFW*-like interface than it is to implement an *MPS_INTERFACE*-like one. This is because the former does not require an SML translator, whereas the latter does. We remark that *MPS_INTERFACE* took approximately 1 person-year to design and implement, whereas *GENNETFW* took less than a fifth of that effort.

This experience provided the incentive to study more closely the technology of *MPS_INTERFACE*-like interfaces. The result of this study can be found in [21], which describes a prototype translator writing system that partially automates the development task for algebraic modeling languages.

*GENNETFW* produces its optimizer-ready input file much faster than *MPS_INTERFACE* does for models of comparable size. This is because *GENNETFW* does no SML translation. The modeler essentially "hard-codes" information about the schema and table structure into the *GENNETFW* control table once and for all (or more precisely, until the schema or table structure changes).

In contrast, *MPS_INTERFACE* translates the schema every time it is invoked, which is time consuming. (We note that the solver interfaces for such languages as AMPL and GAMS work this way also.) In hindsight, it seems wasteful for *MPS_INTERFACE* to translate the schema *ab initio* when only the detailed data have been modified. We believe that the performance of *MPS_INTERFACE*-style interfaces could be made competitive with that of *GENNETFW*-style ones, and now suggest how this might be done.

(d) *Exploit model structure/instantiating data separation.* We can see in retrospect that *MPS_INTERFACE*'s current design does not abide by our first design principle. Probably it would have been better if *MPS_INTERFACE* had been designed to exploit SML's sharp separation between model structure and instantiating data. We could have designed *MPS_INTERFACE* in a way which split processing into two phases: the first could have translated the schema and table structure, storing an intermediate result, and the second could have used the intermediate result and the detailed data to create the solver input file. The key advantage is that the first phase

would not have to be repeated when only the detailed data has changed, thus leading to greater processing efficiency.

For example, the first phase could be analogous to *FcEval*. It could use the schema and table structure to create a program that reads the elemental detail tables and creates an MPS format file for the model/problem specified in the schema. The second phase could be analogous to *Evaluate*.

(e) *Integrate optimizer interfaces with resident evaluator. MPS_INTERFACE* and *FcEval/Evaluate* are not integrated. Much of *MPS_INTERFACE*'s processing consists of evaluating indexed families of expressions, and yet *MPS_INTERFACE* does not share any code with *FcEval*, nor does it use the results of *Evaluate*. We estimate that nearly 75% of *MPS_INTERFACE*'s code duplicates that of *FcEval/Evaluate*! The fact that SML was still evolving while these processes were being implemented has made this duplication particularly costly, because changes in SML's generic rule language require changes to both of these massive processes. This duplication is costly in terms of both development time and modeling environment quality. Note that if the generic rule sublanguage is implemented correctly in any one process with probability $P$, then the probability that $n$ independently developed processes implement it correctly is only $P^n$. Clearly, evaluation should be well integrated with all solver interfaces that require it.

This observation suggests an approach that is at odds with current practice: that optimizer interfaces should be integrated with the resident evaluator. This of course requires the evaluator to be designed so that it is compatible with this expanded mission.

### 5.2.4. *Interfaces to external interactive systems: XtrieveQuery and PrologQuery*

FW/SM incorporates interfaces to the stand-alone interactive systems Xtrieve and Arity Prolog. The overall architecture of these interfaces is similar to that of FW/SM's optimization interfaces, in the sense that communication is via external files. The only differences are: (1) the "solvers" being interfaced are interactive, whereas optimizers normally run transparently, and (2) no "post-solver" translation is done, since these interfaces are designed to be unidirectional.

In general, these two interfaces were straightforward to implement. Our experience inspires two comments for interfaces of this type.

First, it is desirable that the user should be able to refer to all model components by their SML names when interacting with the interfaced system. However, since other systems do not have the same lexical conventions as SML, some mapping must be done from SML names to names in the target system. Moreover, this mapping must be one-to-one and preserve the original SML names insofar as possible. Although this is a relatively minor problem, it does mean that this type of integration may not be perfectly transparent to the user.

Second, the primary purpose of *XtrieveQuery* is to provide functionality that is not available in the core environment, namely relational data querying and

logic-based inferencing. Although these processes certainly work, communication via external files may not be the best way to implement this type of functionality. There is a setup cost to export data from Framework, transform it into the format required by the target system, and then invoke and set up the target system for the user. This cost may be too high when it is desired to frequently interleave queries or inferences with other FW/SM processes. Tighter integration would enhance user productivity, but would require major architectural changes to avoid exorbitant implementation costs.

## 6. Conclusion

We close with comments on two aspects of FW/SM: its underlying design principles, and its delivery platform.

### 6.1. DESIGN PRINCIPLES

Our experiences have solidified our belief in the design principles given in section 1. We summarize these principles and how FW/SM realizes them:

DP1. *Divide processing according to the relative stability of model components.*

- *SMLCheck* checks the formal part of the schema and stores a parsed representation in IRT files for use by other processes. As a result, correctness checks and certain types of parsing need not be repeated until the formal part of the schema changes, thus avoiding redundant processing.
- Evaluation is split into companion processes, *FcEval* and *Evaluate*. *FcEval* compiles the schema and table structure into C code and stores the result. *Evaluate* uses the stored program to evaluate model instances. Recompilation need not be repeated until the formal part of the schema or table structure changes, thus avoiding redundant processing.
- The AUXIL subdirectories and access methods are used to store and retrieve intermediate results. Persistent storage of intermediate results is an essential requirement of DP1.

DP2. *Do not force users to pay the price of performing multiple tasks when they only need to perform one.*

- By comparison with some modeling systems, FW/SM has a large number of separately invokable processes.
- A number of FW/SM processes target the formal part of the schema only, permitting users to make useful analyses without having to develop an entire model instance. For example, users can invoke correctness checks, make logic-based queries, and generate reference documentation, all with respect to general model structure without any interpretation part of the schema and those which target only the detailed data.

- All DP1 examples are also DP2 examples, since the purpose of DP1 is to reduce redundant processing. Generally speaking, DP1 and DP2 tend to be mutually reinforcing.

DP3. *Provide "core services" as a basis for extensibility and quality.*

- The IRT files, together with the AUXIL subdirectories and access methods, are core services available to developers wishing to extend FW/SM.

DP4. *Hide details about the user interface from modeling environment processes.*

- FW/SM's Framework-based user interface makes use of fonts, outlines, and database frames to represent SML models conveniently. Many FW/SM processes, including *SMLCheck*, *FcEval/Evaluate*, *MPS_INTERFACE*, *XtrieveQuery*, and *PrologQuery*, know nothing about this representation. Rather, they read ASCII representations of modeling objects. Details about the user interface are encapsuled in routines that translate objects from the Framework representation to the ASCII one.

Many improvements could be made in our implementation. In particular, FW/SM could have done better at abiding by DP1 and DP3. Some examples of this have already been discussed in conjunction with *MPS_INTERFACE*. Another important example related to DP3 follows.

A capable database management system for handling detailed data is one of the most important core services of any modeling environment. Ideally, there should be only one database system for this purpose. Its physical layer is to the elemental detail tables as the IRT files are to the schema. The challenge is to find a single database system that meets the needs of all modeling environment processes and either has an appropriate user interface or integrates well with a system having one.

FW/SM falls short in this regard. Our selection of Framework's database frames as the foundation for FW/SM's database management facilities proved to be inadequate due to storage and access inefficiencies, even in the context of a research prototype that does not aspire to deal with large models. Consequently, we were forced to integrate an external programmer's database package, namely dbVista, in order to achieve acceptable performance for *Evaluate*.

Moreover, Framework's inability to conveniently express or adequately process multi-table queries cripples its ability to exploit structured modeling's strong relational database compatibility for the purpose of ad hoc query. This was unacceptable: we view ad hoc query facilities to be an essential modeling environment functionality. We therefore elected – after unsuccessful attempts to program the missing capabilities – to interface a second database package, namely Xtrieve, in order to (a) compensate for the Framework limitation just mentioned, and also (b) demonstrate that bridges to popular query interfaces and query engines are feasible. Even making allowances for motivation (b), FW/SM as it stands does not set a good example as regards core services in the database area. Using multiple databases in this way slows down

processing time owing to the extra data export and import, and increases the size of the modeling environment (Xtrieve, dbVista, and Framework must all be on the user's hard disk).

This experience illustrates the importance of building on a suitable software foundation where essential core services are concerned.

### 6.2. DELIVERY PLATFORM

FW/SM is built upon Framework, which embodies numerous well integrated features that are very desirable for modeling environments. These include cut and paste facilities, hierarchical (outline-based) organization of user-created objects, a variety of useful services (e.g. for word processing, table processing, graphing, spreadsheeting, and communicating), a direct manipulation style, and windows. As a group, these features were ahead of their time when Framework was released. The price that we paid to obtain such features for the FW/SM prototype is that Framework must be abandoned – or at least supplemented in a major way – in order to scale up to a commercial quality modeling environment able to handle large volumes of data efficiently.

Fortunately, the desktop computing industry is entering an age in which multitasking operating environments, graphic user interfaces, and powerful database management systems are becoming affordable, popular, and increasingly easy for developers to exploit. Such products as Microsoft's Windows and OS/2, Quarterdeck's DESQview, Borland International's Paradox, and others for personal workstations, make it possible even today to achieve FW/SM's capabilities and more in a modeling environment of commercial quality.

We hope that our experience with FW/SM will enable others to make better evaluations of today's modeling systems and achieve better designs for the modeling systems and environments of tomorrow.

## Appendix: Summary of FW/SM processes

Because this paper frequently refers to specific FW/SM processes, we list them here and summarize their functions. They are described in detail in Geoffrion [15]. This appendix is adapted from appendix S in Geoffrion [14].

Each FW/SM process provides one functionality or coherent group of functionalities, and is invoked by selecting it from a tree-structured menu of processes. We categorize processes as follows: (1) processes which apply to the schema, (2) those which apply to the elemental detail tables, and (3) those which correspond to solvers.

### A.1. SCHEMA (GENERAL STRUCTURE) PROCESSES

*COSMET* Size schema frames and delete extra blank lines

This purely cosmetic process beautifies on-screen displays of the schema.

*DBREF* Translate adjacency/reachability matrices from text to tables

The genus graph adjacency and reachability matrices produced by *REFGEN* (described below) are text displays. *DBREF* converts them to true tables, so that they can be manipulated by Framework's table editor.

*FORMAT* Format the schema

This process assures proper indentation when a schema is printed, and also does some preparatory work for *SMLCheck* (described below).

*INTER_CK* Check the interpretation part of the schema

*SMLCheck* delegates some of its responsibilities with respect to the relatively informal interpretation part of a schema to *INTER_CK*, which checks the syntax of the so-called "defined key phrases", interactively recognizes so-called "referenced key phrases", and checks the rules for proper use of referenced key phrases.

*NETDRAW* Browse the genus graph

*NETDRAW* generates a schema's genus graph as a graphical display that the user can interactively browse so as to better understand a model's general structure.

*REFGEN* Generate reference documentation

*REFGEN* automatically generates various reference documents on the schema useful for communication, debugging, model maintenance and evolution, and other purposes. (Other reference documentation is generated by *EDGEN*, a process described in section A.2.)

*SMLCheck* Check schema syntax and schema properties

SML's specification includes not only a formal context-free grammar describing SML's lexical and syntactic structure, but also an exhaustive collection of so-called schema properties describing the context-sensitive conditions needed for a schema written in this grammar to be consistent with the conceptual Framework given in Geoffrion [12]. *SMLCheck* detects and reports violations of schema properties as well as schema syntax.

## A.2. ELEMENTAL DETAIL TABLE (INSTANTIATING DATA) PROCESSES

*EDGEN* Generate skeletal elemental detail tables

*EDGEN* generates empty elemental detail tables, with a flexible option to join contiguous tables. It also automatically generates two additional reference documents beyond those created by *REFGEN*. The elemental detail tables are relations in the sense of relational algebra, and FW/SM has a table editor that permits their direct manipulation.

*FcEval* Compile generic rules into C

The *EDGEN* process does not create a mechanism for doing evaluation, that is, for filling in values for function and test elements. *FcEval* provides this capability by compiling generic rules into C code that can be run by the *EVALUATE* process. This code need never be regenerated so long as the schema stays the same, thus giving the benefits of "warm restart" for multiple evaluations.

*L/E UTILITIES* Utilities for loading/editing tables

This process, which comprises several subprocesses, supports various set and relational operations commonly needed when building and maintaining elemental detail tables.

*MATRIX* Build matrix from table with two key fields

Elemental detail tables with two key fields for attribute, function, or test genera amount to a linearized representation of a matrix. *MATRIX* transforms such an elemental detail table into the corresponding matrix.

*TABLE RULES* Check table content rules

There is a collection of so-called table content rules that are exhaustive in a strong sense for guaranteeing the internal consistency of the elemental detail table (Geoffrion [11]). This process is a partial implementation of these rules.

*ID_DICT* Build identifier dictionary

*ID_DICT* creates a database of all identifiers appearing in the defining elemental detail tables associated with self-indexed genera. Information on defining genera and identifier interpretations is included.

### A.3. SOLVER PROCESSES

*EVALUATE* Perform evaluation

*EVALUATE* performs generic rule evaluation using compiled C code, provided *FcEval* has been invoked previously. Results are put into the elemental detail tables where they are readily accessible to Framework's table editor and to other processes. Immediate evaluation capability is all but indispensable for debugging, answering "What If?" questions, for doing many kinds of analysis of the model and of derived results, and for generating reports.

*GENNETFW* Invoke generalized network flow solver

This is a control table interface that builds a suitable problem file for generalized network flows and invokes the GENNET optimizer.

*MPS_INTERFACE* Invoke linear/integer programming solver

This is a completely automatic interface that builds a suitable MPS problem file for linear and integer programming and invokes the LINDO optimizer (Schrage [26]).

*PrologQuery* Prolog-based schema query

This process provides a fully automatic interface to a commercial Prolog system (Arity [2]) in such a way that logic-based inferencing can be done with respect to certain basic aspects of the schema. The process translates Schema information into Prolog predicates (facts and rules), exports these along with certain other "stock" predicates to the Prolog system, invokes the Prolog system with proper initial conditions, and deposits the user in the command-driven Prolog environment. The full power of Arity Prolog is available to the user for answering queries (drawing logical inferences) based on the available schema information.

*XtrieveQuery* Menu-based elemental detail table query

This process exports all elemental detail tables to Xtrieve, a commercial quasi-relational query interface and processor (Novell [23]), invokes Xtrieve with proper initial conditions, and deposits the user in the menu-driven Xtrieve environment. The full power of Xtrieve is available to the user for answering queries concerning data and results, and for generating reports.

## Acknowledgements

Partially supported by the National Science Foundation, the Office of Naval Research, Ketron Management Science, and Shell Development Company. The views in this report are those of the authors and not of the sponsors. We appreciate the thoughtful comments provided by Chris Jones and several anonymous referees.

## References

[1] A.V. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison–Wesley, Reading, MA, 1986).

[2] Arity Prolog Version 5.0, Arity Corporation, 30 Domino Drive, Concord, MA 01742 (1987).

[3] *Framework IV*, Ashton-Tate, 20101 Hamilton Ave., Torrance, CA 90502 (1991).

[4] A. Brooke, D. Kendrick and A. Meeraus, *GAMS:A User's Guide* (Scientific Press, Redwood City, CA, 1988).

[5] G.G. Brown and R.D. McBride, Solving generalized networks, Manag. Sci. 30(1984)1497–1523.

[6] D.R. Dolk, Model management and structured modeling: The role of an information resource dictionary system, Comm. ACM 31(1988)704–718.

[7] A.S. Drud, Interfaces between modeling systems and solution algorithms, in: *Mathematical Models for Decision Support*, ed. G. Mitra (Springer, Berlin, 1988).

[8] S.I. Feldman, Make: A program for maintaining computer programs, Software: Practice and Experience 9(1979)255–265.

[9] R. Fourer, D.M. Gay and B.W. Kernighan, AMPL: A mathematical programming language, Manag. Sci. 36(1990)519–554.
[10] A.M. Geoffrion, An introduction to structured modeling, Manag. Sci. 33(1987)547–588.
[11] A.M. Geoffrion, SML: A model definition language for structured modeling, Working Paper 360, Western Management Science Institute, UCLA (May, 1988), Revised 11/89, 8/90.
[12] A.M. Geoffrion, The formal aspects of structured modeling, Oper. Res. 37(1989)30–51.
[13] A.M. Geoffrion, Computer-based modeling environments, Eur. J. Oper. Res. 41(1989)33–43.
[14] A.M. Geoffrion, The SML language for structured modeling, Working Paper 378, Western Management Science Institute, UCLA (August, 1990). Two-paper extract appeared in Oper. Res. 40(1992)38–75.
[15] A.M. Geoffrion, FW/SM: A prototype structured modeling environment, Manag. Sci. 37(1991) 1513–1538.
[16] A.M. Geoffrion, S. Maturana, L. Neustadter, Y. Tsai and F. Vicuña, Technical documentation for FW/SM, Anderson Graduate School of Management, UCLA (April, 1990).
[17] A.M. Geoffrion, S. Maturana, L. Neustadter, Y. Tsai and F. Vicuña, User documentation for FW/SM release X92-04, Anderson Graduate School of Management, UCLA (April, 1992).
[18] S.C. Johnson, YACC: Yet another compiler-compiler, UNIX prog. Manual Vol. 2B, Bell Laboratories, Murray Hill, NJ (July, 1978).
[19] M.L. Lenard, Representing models as data, J. Manag. Inf. Syst. 2(1986)34–48.
[20] M.E. Lesk, Lex: A lexical analyzer, Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ (1975).
[21] S.V. Maturana, A translator writing system for algebraic languages, Ph.D. Dissertation, Anderson Graduate School of Management, UCLA (1990).
[22] *QuickC Compiler Version 2.0*, Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717 (1989).
[23] *Xtrieve: Interactive Query Manual*, Novell Development Products Division, 6034 W. Courtyard, Suite 220, Austin, TX (1988).
[24] *db_FILE: File Manager C-Language User's Guide*, RAIMA Corporation, 3245 146th Place SE, Bellevue, WA 98007 (1988).
[25] T.W. Reps and T. Teitelbaum, *The Synthesizer Generator* (Springer, New York, 1989).
[26] L. Schrage, *User's Manual for Linear, Integer, and Quadratic Programming with LINDO* (Scientific Press, Redwood City, CA, 1987).
[27] R.N. Taylor, R.W. Selby, M. Young, F.C. Belz, L.A. Clarke, J.C. Wileden, L. Osterweil and A.L. Wolf, Foundations for the Arcadia environment architecture, SIGPLAN Notices 24(1989)1–13.
[28] Y. Tsai, Structured modeling query language, Ph.D. Dissertation, Anderson Graduate School of Management, UCLA (1990).
[29] L. Tung, Relational representations of structured modeling, Dept. of Decision and Information Systems, Arizona State University (February, 1990).
[30] F. Vicuña, Semantic formalization in mathematical modeling languages, Ph.D. Dissertation, Computer Science Department, UCLA (June 1990).
[31] F. Vicuña, T. Aiken and A.M. Geoffrion, Technical documentation for SMLCheck, Anderson Graduate School of Management, UCLA (November, 1990).