Calhoun: The NPS Institutional Archive

DSpace Repository

Faculty and Researchers | Faculty and Researchers' Publications

1992-03

Model Integration and Modeling Languages: A Process Perspective

Kottemann, Jeffrey E.; Dolk, Daniel R.

INFORMS

Kottemann, Jeffrey E., and Daniel R. Dolk. "Model integration and modeling languages: A process perspective." Information Systems Research 3.1 (1992): 1-16.
http://hdl.handle.net/10945/70155

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

Model Integration and Modeling Languages: A Process Perspective

Jeffrey E. Kottemann
School of Business Administration
The University of Michigan
Ann Arbor, Michigan 48109-1234

Daniel R. Dolk
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93943

Development of large-scale models often involves—or, certainly could benefit from—linking existing models. This process is termed *model integration* and involves two related aspects: (1) the coupling of model representations, and (2) the coupling of the processes for evaluating, or executing, instances of these representations. Given this distinction, we overview model integration capabilities in existing executable modeling languages, discuss current theoretical approaches to model integration, and identify the limiting assumptions implicitly made in both cases. In particular, current approaches assume away issues of dynamic variable correspondence and synchronization in composite model execution. We then propose a process-oriented conceptualization and associated constructs that overcome these limiting assumptions. The constructs allow model components to be used as building blocks for more elaborate composite models in ways unforeseen when the components were originally developed. While we do not prove the sufficiency of the constructs over the set of all model types and integration configurations, we present several examples of model integration from various domains to demonstrate the utility of the approach.

Model management—Model integration—Modeling systems—Modeling languages

1. Introduction

A fundamental assumption underlying model management is that models are sharable resources. A model management system (MMS) facilitates sharing by supporting a uniform representation for, and manipulation of, models. One result of increased model sharing is that models may eventually be used for purposes not originally envisioned by their creators. Specifically, models may be co-opted as building blocks in the creation of larger, more complex models. This process is known as model integration.

Model integration involves the construction of a composite model built from two or more existing model components. For example, a national energy model may be

1047-7047/92/0301/0001/$01.25
Copyright © 1992, The Institute of Management Sciences

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

built by combining a number of regional energy forecasting models on the demand side with an overall transportation model for energy allocation on the supply side, and then imposing the appropriate equilibrium conditions. Model integration has long been a desirable goal but one hampered by incompatible representations imposed by different software systems. An MMS based on a single model representation scheme significantly enhances opportunities for successful model integration, but major problems must still be confronted.

(1) At the logical level, integrating model schemas from different domains raises issues of semantics and dimensionality when trying to identify common elements among two or more models. For example, financial and marketing models may represent revenues as net present $ and regular $, respectively. Failure to recognize this difference will lead to improper coupling of the two models.

(2) At the manipulation level, specifying procedures to evaluate and solve composite models becomes a nontrivial task. For example, if one joins an integer programming model with a nonlinear programming model, what solution procedure(s) should be used? Does one retain the nonlinear algorithm or devise an entirely new procedure using other algorithms?

Our objective in this paper is to define in a broad sense the characteristics of an executable modeling language (EML) necessary to support model integration, with particular emphasis on the manipulation level. Both the logical and, in particular, the manipulation dimensions of model integration have implications for EMLs which have been largely overlooked in the literature. In §2 we briefly survey existing EMLs to show that current EMLs are either largely representation-oriented with few capabilities for model manipulation as in the case of optimization, or essentially programming languages as in the case of discrete event simulation. In §3 we identify a set of key assumptions which have been (implicitly) made in the model management literature regarding model integration. In following sections we show how relaxing these assumptions leads to various process-oriented constructs needed to coordinate execution of composite models. Our conceptualization draws on formalisms in communicating sequential processes (CSP) (Hoare 1985) and discrete event simulation (DES) (Zeigler 1984).

2. Executable Modeling Languages

Modeling languages come in many different forms which differ according to the domain of applications they serve. In the optimization world, for example, languages are largely representation-oriented. At the other end of the spectrum, languages for DES applications are essentially procedural programming languages (i.e., manipulation-oriented). The wide range of modeling languages which exist across different domains presents unique problems with regard to model integration as the example below demonstrates.

Consider a firm which has developed the following models as shown in Figure 2-1 (this example has been adapted from (Blanning 1986). After the model user posits an initial price, the following model components are evaluated:

(1) given the currently posited price, an econometric marketing model forecasts demand for a product for the next fiscal year,

(2) a discrete event simulation production model estimates the required time and expense to manufacture enough of the product to meet demand,

(3) a transportation model determines the minimal cost of distributing the product to customers,

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

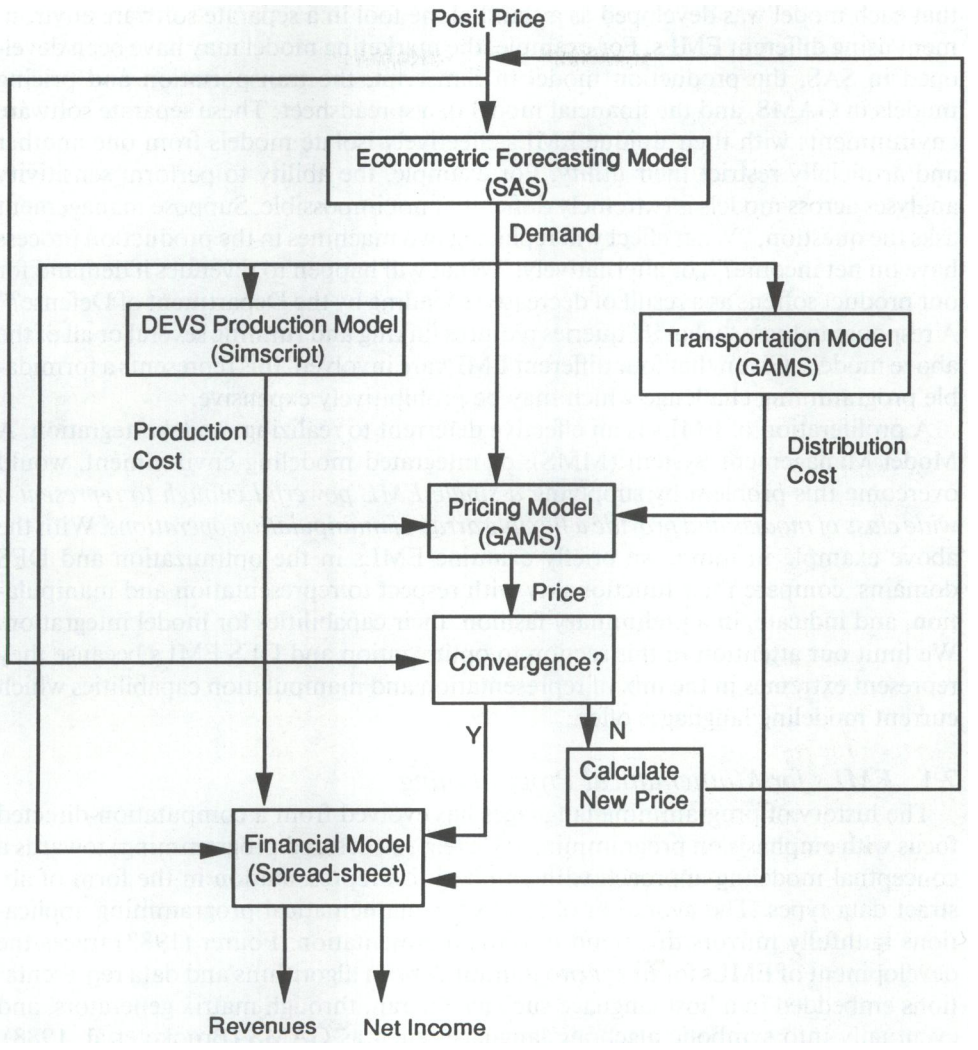

FIGURE 2-1. An Integrated Model Example (Adapted from Blanning 1986).

(4) a pricing model calculates a price for a product given demand, and production and distribution expenses,

(5) the newly calculated price and the last posited price are checked for convergence. If suitable convergence is not attained the above series of evaluations is performed again, where the posited price now takes on the value of the newly calculated price,

(6) if suitable convergence is attained, a financial model determines the revenues and net income from sales of the product.

Several aspects of this scenario merit attention. Note from Figure 2-1 that the models are conceptually interconnected in the sense that their outputs could serve as inputs to succeeding models. However, the exact nature of the interconnection is dependent on the outcomes of evaluating model component #5. Further, it is likely

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

that each model was developed as a stand-alone tool in a separate software environment using different EMLs. For example, the marketing model may have been developed in SAS, the production model in Simscript, the transportation and pricing models in GAMS, and the financial model in a spreadsheet. These separate software environments with their unique EMLs effectively isolate models from one another and artificially restrict their utility. For example, the ability to perform sensitivity analyses across models is extremely difficult, if not impossible. Suppose management asks the question, "What effect will replacing two machines in the production process have on net income?", or alternatively, "What will happen to revenues if demand for our product softens as a result of decreased spending by the Department of Defense?" A response to these "what if" queries requires linking and running several or all of the above models. Given that four different EMLs are involved, this represents a formidable programming challenge which may be prohibitively expensive.

A proliferation of EMLs is an effective deterrent to realizing model integration. A Model Management System (MMS), or integrated modeling environment, would overcome this problem by supplying *a single EML powerful enough to represent a wide class of models and provide a flexible array of manipulation operations.* With the above example in mind, we briefly examine EMLs in the optimization and DES domains, compare their functionality with respect to representation and manipulation, and indicate, in a preliminary fashion, their capabilities for model integration. We limit our attention in this section to optimization and DES EMLs because they represent extremes in the mix of representation and manipulation capabilities which current modeling languages offer.

2.1. *EMLs for Mathematical Programming*

The history of programming languages has evolved from a computation-directed focus with emphasis on programming style (e.g., structured programming) towards a conceptual modeling approach with emphasis on representation in the form of abstract data types. The evolution of EMLs for mathematical programming applications faithfully mirrors this trend towards representation. Fourer (1983) traces the development of EMLs for linear programming from algorithms and data representations embedded in a host language such as Fortran, through matrix generators, and eventually into symbolic algebraic languages such as GAMS (Brooke et al. 1988), AMPL (Fourer et al. 1990), and SML (Geoffrion 1987, 1990).

Algebraic languages are largely representational in nature, that is, they are concerned with describing a model but not with evaluating or solving it. Manipulation is done by constructing the proper interface with external routines which perform the required operations. These EMLs have elaborate syntax for problem description but rudimentary capabilities for manipulation. In general, the manipulation aspect of these languages does little more than to specify which solution algorithm to apply to model representations. Further, most systems have very restricted or nonexistent features for model integration. GAMS is an exception in that it does provide rudimentary integration capabilities as part of its modeling language. In particular, GAMS allows the model builder to define a sequence of models, and their appropriate solvers, to be executed in series. GAMS provides a preliminary indication of what's required to perform model integration, namely a language which is a level removed from the details of any single model and which therefore can manipulate models as configurable components. As we show in later sections, however, the requirements

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

for a language which fully realizes model integration are more extensive than what GAMS, or any EML we're aware of, supports. Moreover, an EML for model integration should facilitate integration involving models from different paradigms such as mathematical programming and DES.

2.2. *EMLs for Discrete Event Simulation*

EMLs in the discrete event simulation (DES) world are full-fledged programming languages such as Simscript (SIMSCRIPT 1983) and SLAM (Pritsker and Pegden 1979), with elaborate representation and manipulation capabilities. These languages differ from conventional programming languages in that they are oriented towards modeling dynamic, stochastic phenomena. In this environment events or processes are activated during program execution according to a schedule which depends on an underlying system clock. As a result, program flow does not occur serially as in conventional third generation programming languages such as Fortran or Pascal, but rather as independent processes which are triggered by some external condition(s).

The separation between model definition and manipulation in DES languages is very indistinct. We cannot neatly define a boundary between the representation and the solution algorithm as in the case of optimization models. The relationship between solvers and models is one-to-many in the mathematical programming world whereas it is closer to one-to-one in DES applications. A simplex algorithm, for example, can be used to solve a virtually unlimited number of different linear programming model instances, each with a closed and complete mathematical description. Simulation models, on the other hand, are formulated to solve one particular problem, and are not especially portable to other applications. This shortcoming of existing DES languages has led DES researchers to explore alternatives for strengthening the separation between DES model definition and manipulation (see, e.g., Zeigler 1984, and the collections of papers in Widman et al. 1989, and Oren et al. 1984).

3. Dimensions of Model Integration

In this section we propose a useful way to classify the dimensions of, and approaches to, model integration. A model component consists of a model schema (in the spirit of structured modeling (Geoffrion 1987)) plus a set of one or more processes that evaluate the composite model. There are two critical aspects that must be defined when specifying how components are to be integrated to form a composite model:

(1) Variable Correspondence: the input/output relationship between model component variables specifying which outputs from one model component serve as inputs to other model components;

(2) Synchronization: the order in which model components must be manipulated, or executed, and the timing of dynamic interaction of model components.

Take, for example, a composite model where econometric forecasting models (M_1 and M_2) are used to generate projections for the demand and supply of an energy commodity, and these projections are then used as input to a linear programming model (M_3) to determine distribution of the commodity (Figure 3-1). In this simple case, models M_1 and M_2 execute to termination (perhaps in parallel), then the demand and supply variables are transmitted, and M_3 begins execution. The composite model consists of a partial ordering of model components which forms a simple,

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

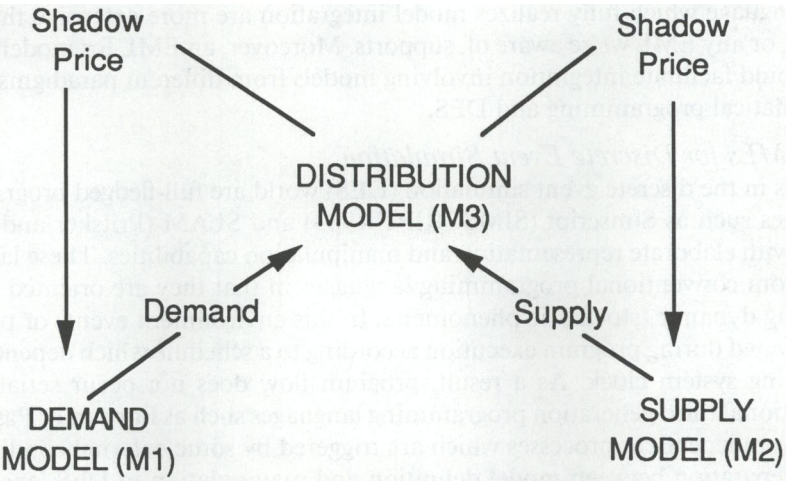

FIGURE 3-1. Example of Model Integration (Adapted from Shapiro 1978).

well-behaved precedence relationship. Thus synchronization is trivial in this example.

Now consider an extension to the model which introduces equilibrium conditions (e.g., Shapiro 1978). In this case, M_3 feeds back into M_1 and M_2 and iterations of this cycle are required in order to satisfy equilibrium conditions. This requires a more elaborate ordering of processes (specifically an iteration operator) to solve the composite model, although synchronization is still relatively simple.

Nontrivial synchronization would be required if all three model components had to run in parallel and communicate with one another during their execution. Thus, components M_1 and M_2 might run part of their processes (one internal iteration, e.g.), then output variables to M_3, which would run all or part of its process, return output to M_1 and/or M_2 for another internal iteration, and so on until convergence. This is clearly more tightly coupled than the previous scenario.

The above series of examples illustrates a continuum of model integration which we label with the terms *consolidation, pipelining,* and *splicing.* As discussed below, this proposed continuum is based upon the complexity and dynamism of variable correspondence and synchronization involved in specifying an evaluation process for a composite model.

Consolidation is the simplest case, indeed a trivial case in terms of process specification. In consolidation, multiple model representations are "joined" and their *evaluation is performed by a single process.* For example, two regional network models can be combined into an overall national network model, and the resulting composite model can be evaluated using a single network model evaluation process. (See Geoffrion 1989a, b and Bradley and Clemence 1988 for other examples of consolidation.) Because evaluation is performed by a single process, variable correspondence is static and determined prior to model execution and interprocess synchronization is not an issue.

Pipelining involves models that are separate, communicating processes, but

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

synchronization can be defined by a *predefined and static partial ordering.* Pipelining is based on the following synchronization assumption:

Composite models form a directed, acyclic graph that defines a partial ordering, such that model M_i may run to termination before M_j is activated for all $i < j$ given by the partial ordering.

Variable correspondence is assumed to be static and bound by the definition of the partial ordering. Similarly, synchronization is simple and static because the ordering of execution is, by definition, not affected by the run-time behavior of model components. The only synchronization consideration is to insure that processes execute as predefined by the partial ordering and that processes wait until they are provided input data.

Consolidation and pipelining are the dominant views of model integration taken in the model management literature (e.g., Blanning 1985, 1986; Miller and Katz 1986; Muhanna and Pick 1988; Sagie 1986; Sprague 1980, Liang 1986; Roy et al. 1986). However, it is important to realize that these constitute two quite different approaches to model integration. Consolidation is effectively the logical integration of homogeneous models, where by "homogeneous" we mean models which have the same solver. This entails primarily the integration of model schemas as detailed in Geoffrion (1989) but little, if any, process integration. Pipelining, on the other hand, is relevant to the multi-paradigmatic situation, where the models and their associated solvers are heterogeneous. In this case, we are primarily interested in process, or solver, integration. The complexities inherent in process integration have been largely overlooked in the literature, or else simplified to the point where the problems of dynamic variable correspondence and synchronization are assumed away.

The basic variable correspondence and synchronization assumptions which characterize current model management research oversimplify the problem of model integration in many practical situations. Many situations require definition of more complex and dynamic variable correspondences and synchronization protocols than the above perspectives afford (see, e.g., Shapiro 1978, Dolk and Kridel in press). Specifically, the following limiting assumptions are implied by the above basic perspectives:

A1. Variable correspondences are static, i.e., they are bound prior to, and do not change during, the execution of the composite model.

A2. Run-time behavior does not affect synchronization. Further, components are executed unconditionally, and there are no interleaved execution patterns due to cycles.

A3. Only those components' variables defined *a priori* as output variables can be referenced, *and* then, only when the component normally makes them available.

A1, 2, and 3 essentially serve to assume away the dynamic dimension of model integration. Although this may be warranted in many cases, even the "simple" process of calculating batch means violates A1. As shown in the following sections, model evaluation strategies may dictate a *Splicing* of two or more independent model evaluation processes that violates any or all of these assumptions, even in situations where the model components considered in isolation are relatively static. In such

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

situations, variable correspondence and synchronization may both be dynamic in nature. Further, relaxing assumption A3 (discussed in §5) makes it possible to use existing model components in unforeseen ways, thereby dramatically improving the potential reusability of model components.

4. Composite Models as Communicating Processes—Relaxing Assumptions A1 and A2

Recalling from the previous section, current approaches to model integration make the assumption that variable correspondences are static (A1) and that run-time behavior does not affect synchronization (A2). While these are sometimes valid assumptions, the model integration example given in Figure 2-1 violates both. First, the run-time results of the model component that tests for convergence dynamically affect variable correspondences. Second, components are not evaluated, or executed, unconditionally and there are (sub)cycles in the execution pattern. In this section, we describe MMS formalisms and process-oriented constructs that permit relaxation of the above two assumptions. Following that, the constructs are illustrated using the example of Figure 2-1.

One relatively straightforward way to accommodate dynamic variable correspondence and synchronization is to, first, consider model components in the model base as processes that communicate by sending and receiving messages, and, second, to recognize that the messages sent and received can be either data or control/status variable values. By doing the first, variable correspondences can be rerouted by an overlying, executive process; by doing the second, the executional state, or dynamic behavior, of the model components can be monitored and controlled.

For the sake of parsimony, all model components in the model base are assumed to be functionally equivalent to the following canonical form:

Init:	procedure(s) performed when the component is invoked. Wait for Start message.
Body:	the main task that the model component performs.
Fini:	procedure(s) performed when the component finishes on its own or when it is sent a Finish message.

For example, a model component that calculates the mean of a series of data points would receive messages in the Body and send messages (results) in the Fini.

Init:	count := 0; sum := 0; {Wait for Start signal}
Body:	Receive Data; sum ++ Data; count ++1;
Fini:	Send sum/count

By sending control/status messages, model components communicate their behavior via state transition messages; by receiving control/status messages, model components allow their behavior to be manipulated dynamically. A complementary form of the former is to consider that each component maintains an encapsulated *set* of *status variable values,* **SV,** which can be referenced to determine the current operational state of the component. This permits checking the state of a model component (rather than being notified of a state transition via an explicit message from a component). For example, the statement **If Finish in mean.sv,** tests whether the component model, Mean, has completed execution, and **Send mean.Finish,** will cause the mean components to execute its Fini procedure.

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

Given the above canonical form for model components, a model developer can enlist some number of model components from a model base and specify their model integration scheme (MI schema). The MI schema needs to define dynamic procedural interactions between the model components. Therefore, the MI schema is itself a procedure—termed the Model Integration Control Procedure, or the MICP.[1] All messages to and from model components pass through the MICP. Thus the *MICP serves as a message router.* Also, since the MICP can examine messages it receives and since model components wait, or suspend execution, when they reach a Receive statement, the *MICP can act as a message filter and can control the interleaving of the active stages of model components.* For a simple example, take a MICP designed to calculate batch means. To do so it must couple a data producing component, **Sample,** to multiple instantiations of a mean calculation component, **Mean:**

```
Start Sample
Do While Sample not in state Finish (i.e., Finish Not in Sample.SV)
  Start Mean
  Do batch
    Receive Sample.Point; Send Mean.Data.Point;
  Doend batch
  Send Mean.Finish; Receive Mean.Result
Doend
```

This simple example of calculating batch means illustrates the benefits of viewing model integration in terms of (1) model components that are communicating processes which send and receive messages, (2) the distinguished role that status/control variables play when variable correspondence and synchronization are dynamic in nature, and (3) the value of structured programming constructs. Next we demonstrate these constructs using an example adopted from Blanning (1986).

The integrated model of Figure 2-1, when viewed as processes that interact via a MICP, is:

```
  Get PositPrice;
  Repeat Until DONE
    Send Forecasting.PositPrice; Receive Forecasting.Demand;
    Send Production.Demand; Receive Production.PCost;
    Send Transportation.Demand; Receive Transportation.TCost;
    Send Pricing.(Demand,PCost,TCost); Receive Pricing.Price;
    Send TestConverg.(PositPrice,Price);
    If Fail in TestConverg.SV
      PositPrice := (PositPrice + Price)/2
    Else Send Financial.(Price,Demand,PCost,TCost);
      Receive Financial.(Revenues,Net Income);
      DONE
```

Along with the batch mean calculation example given earlier, Blanning's (1986) example suggests that many common model integration situations involve the

[1] The degree to which such procedures can be detected automatically is briefly addressed in the discussion section.

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

dynamic splicing of model components, and that the current assumptions underlying pipelining approaches are invalid in such situations. One way to overcome these assumptions is to view model components as processes that communicate data and status indicators under the supervision of a MICP. The MICP in such situations must dynamically control the interleaved pattern of model component activation. The examples also demonstrate that it is useful to make the basic constructs of structured programming—sequence, selection, and iteration—available to MICPs in order to support the specification of dynamic model integration schemas (also see Bonczek et al. 1986).

5. Dynamic, Unanticipated Interaction—Relaxing Assumption A3

Thus far, the MICP is only able to examine SV values and those variables explicitly sent as messages from model components. Further, the MICP's control of composite models' active phases has been based on the predefined behavior of the model components themselves. That is, the following assumption has yet to be relaxed:

A3. Only those components' variables defined *a priori* as output variables can be referenced, *and* then, only when the component normally makes them available.

As shown in this section, providing a MICP mechanism that relaxes assumption A3 both increases the variety of model component types that can be accommodated by a MMS and improves the potential for reusability of model components.

Relaxing assumption A3 introduces two MMS requirements. First, the MICP must be able to reference all a model component's variables—including those not normally sent as messages. Second, the MICP must have a mechanism to dictate when it wishes to examine a component's variables—other than those times predefined in the component. The latter, in particular, necessitates use of *demons.* A demon is a process that activates when certain conditions arise. Examples of three forms of demons useful in the context of multi-paradigmatic model integration—that is, where models from different modeling paradigms such as math programming and simulation are involved—are given below. Following discussion of these forms, we present an example which shows the advantages of demons in multi-paradigmatic model integration.

(1) *When ANY CHANGE in M.V*

Say the MICP is to calculate mean number in queue in a simulation component M and that queue statistics are not predefined as messages sent by M. A demon can be used to extract the appropriate queue data from M at the appropriate times as follows. First, the demon would be specified with the precondition *When Any Change in M.number-in-queue.* This precondition means that the demon will be activated whenever there is a change in M's variable number-in-queue. The body of the demon procedure would in turn *Send Mean.Data.*(*M.number-in-queue*) each time it was invoked, thus using the model component Mean to perform the ongoing calculation.

(2) *When M (Boolean Expression)*

Here, the demon is invoked not for any change in a (set of) variable(s), but rather only if the change satisfies a given condition. Take, for example, a MICP to calculate batch means in which sample sizes of 30 are to be sent to instances of a mean calculation component. A demon can be defined with the precondition *When*

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

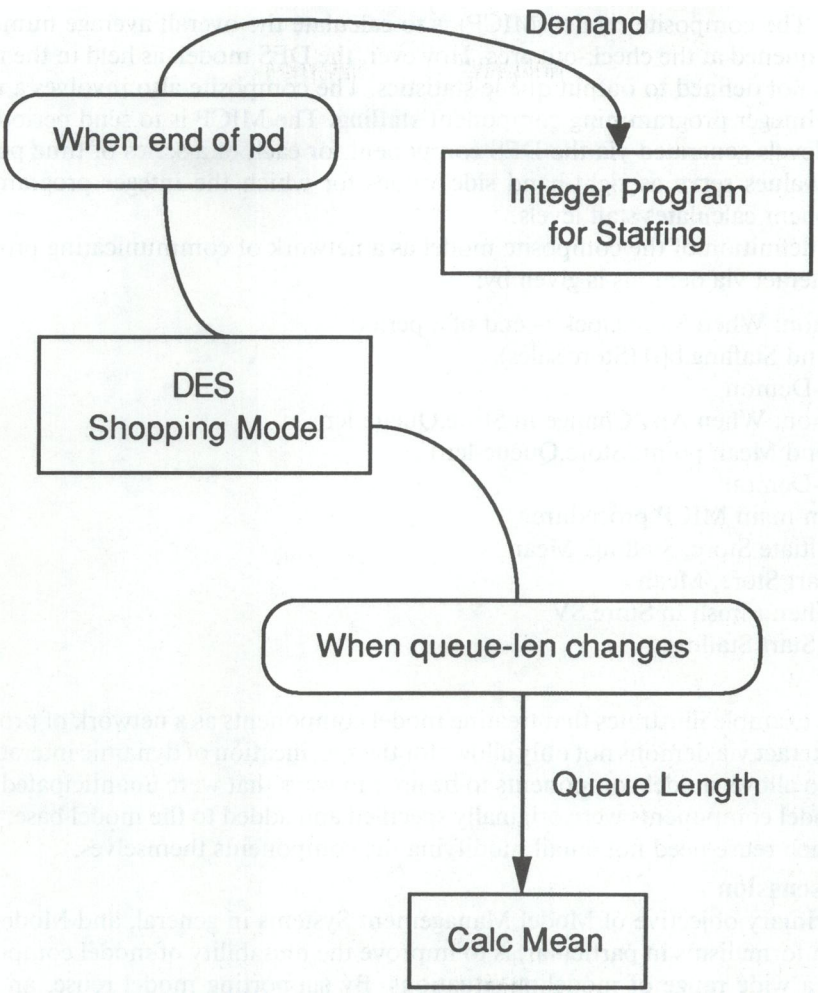

FIGURE 5-1. Schematic for §5 Example.

Mean.Count = 30. The demon procedure, when activated, would then receive the result (after sending a Finish message to Mean) and initiate a new Mean component.

(3) *When M Attains State S—(i.e., When SV-Value Enters M.SV set)*

In many cases, it is more natural to express a demon's condition as a function of model state names. For example, if a MICP needs control after an LP has reached a first feasible solution, a demon with the precondition *When M attains F-F-S* (or equivalently as *When F-F-S enters M.SV*) could be defined.[2]

An Example

The following example involves Discrete Event Simulation (DES), statistical, and Math programming components (see Figure 5-1). The DES component is a **store**

[2] The tedious alternative would involve a lengthy boolean condition in which all the LP's artificial variables must be tested.

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

model. The composite model (MICP) is to calculate the overall average number of people queued at the check-out area. However, the DES model, as held in the model base, is not defined to output queue statistics. The composite also involves a multi-period integer programming component **staffing.** The MICP is to send periodic demand levels generated via the DES component for each of a series of time periods. These values serve as right-hand side values for which the integer programming component calculates staff levels.

The definition of the composite model as a network of communicating processes that interact via demons is given by:

```
Demon: When Store.clock = end of a period
  Send Staffing.b[i].(Store.sales)
End-Demon
Demon: When Any Change in Store.Queue-len
  Send Mean.point.(Store.Queue-len)
End-Demon
Begin main MICP procedure
  Initiate Store, Staffing, Mean
  Start Store, Mean
  When Finish in Store.SV
    Start Staffing
End
```

This example illustrates that treating model components as a network of processes that interact via demons not only allows for the specification of dynamic interactions, but also allows model components to be used in ways that were unanticipated when the model components were originally specified and added to the model base; moreover such reuse need not entail modifying the components themselves.

6. Discussion

A primary objective of Model Management Systems in general, and Model Integration formalisms in particular, is to improve the reusability of model components across a wide range of modeling situations. By supporting model reuse, an MMS helps improve both the reliability of composite models—since the individual components are assumed to have been previously verified—and the rapidity with which large-scale, multi-paradigmatic, composite models can be constructed. In §3, we identify the assumptions underlying current MI approaches. We also discuss how these assumptions limit the range of modeling situations that can be supported, and, by implication, how these assumptions limit the potential reuse of existing modeling components. In §4 and §5, we present a small set of MI "primitives" that relax the assumptions outlined in §3 and that consequently expand the range of model integration situations that can be supported by a MMS. These constructs are summarized in Table 6-1. Their utility is summarized below.

Model integration involves both the coupling of model component representations and the coupling of processes that manipulate the components. In this paper, we pay particular attention to defining the dynamic behavior of composite models during model evaluation and propose constructs intended to support model integration in cases where dynamic variable correspondence and synchronization are required. Below, we review the motivation for these constructs and then briefly discuss when and how a MICP's (sub)procedures might be detected and formulated automatically.

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

TABLE 6-1
Process Integration Constructs

Construct	Comment
Message Passing through the MICP	Provides the foundation to support dynamic variable correspondence and process synchronization
Conditional and Iteration Constructs	Allows for the specification of dynamic variable correspondence and process synchronization in the cases of **Sequence Determinant** control
Demons	Allows for the specification of dynamic variable correspondence and process synchronization in the cases of **Sequence Indeterminant** control

The first requisite for model integration in dynamic cases is the adoption of an appropriate conceptualization of model components and of a mechanism to govern their interactions. Here, model components are conceptualized as processes that send and receive data and status/control messages under the control of a Model Integration Command Procedure. Since the MICP can monitor and control processes and messages, this conceptualization enables dynamic variable correspondence and synchronization. In order to better facilitate such control, the component models are also conceptualized as containing an embedded set of status variables that indicate the current operational state of the components. Thus, the MICP can monitor model states as well as state transitions.

In considering variable correspondence, current "pipelining" approaches (see §3) assume a static precedence relationship. Thus, the issue of "when does model B receive variable (message) X from model A?" is answered simply: "whenever model A sends it." However, the examples of §4 indicate that such an assumption is unfounded in situations ranging from the "simple" calculation of batch means to more complex situations such as product pricing. In addition to allowing the MICP to monitor the states and state transitions of model components, allowing conditional (IF) and iteration (DO) constructs enables the MICP to dynamically control processes by filtering and rerouting messages.

The above constructs relax the assumptions of static variable correspondence and the conceptualization of a composite model as a simple deterministic precedence ordering. However, they still suffer limitations under a third assumption: Procedural conditionals (e.g., IF) are suited only for what might be termed "sequence determinate" control, and their use assumes that one knows when, in the sequence of model component evaluation, the condition should be checked. Take the example of §4 (graphically depicted in Figure 2-1). Checking for convergence (via the IF statement) is sequence determinate because one knows exactly when the condition should be checked in the sequence describing the order of model component evaluation. But, what if one cannot predetermine when a condition should be checked?

In order to widen the range of modeling situations that can be supported by a MMS for MI, and to further improve the potential for model reusability, it is useful to introduce a less sequence determinate conditional: Demons. As shown in §5, employing demons and allowing them to monitor any and all variables in component

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

models further enables the integration of components from a variety of modeling paradigms and allows these components to be used in ways unforeseen when the components were originally developed. Also, since demons are relatively nonprocedural, their use should simplify the definition of composite models relative to use of traditional structured programming constructs.

It is well accepted that nonprocedural constructs have advantages over procedural constructs *vis-a-vis* the burden on end-users—in this case, modelers. This brings up the question: To what extent can MICP (sub)procedures be detected and formulated automatically? While a thorough analysis of this issue is beyond the scope of this paper, it is useful to tentatively explore how automatic procedure detection and formulation might be accomplished in situations where dynamic variable correspondence and synchronization are involved.

To explore this issue, consider the example of §4 (Figure 2-1). The primary dynamic, procedural aspect of this composite model revolves around the convergence checking component. The behavior and output of convergence checking leads to a composite model structure involving both dynamic variable correspondence and an evaluation cycle. What might permit a MMS to detect this substructure and to generate the appropriate MICP procedure automatically? What "knowledge" would it need?

One way to view this problem is in terms of model typing. In addition to typing models by their logical function—e.g., demand forecasting, transportation, etc.—models might also be typed in accordance with their overall procedural function. For example, the convergence checking component could be typed as a "test model," where models of this type are assumed to serve a procedural function that dictates whether their output (e.g., price) should be routed forward or backward. If the MMS is equipped with knowledge regarding the structural implications of such a model type, it might be able to generate the corresponding procedural structure automatically. Determining the necessary set of model types relating to overall procedural function and the development of MMS that can use such typing information to automatically generate MICP structures is an exciting direction for future model management research.

In addition to the primitives shown in Table 6-1, there are several other desirable traits that a MMS for model integration should have:

(1) support of a robust typing and inheritance scheme for variable correspondence and model typing (e.g., Bradley and Clemence 1988);

(2) complementarity with an existing model representation language such as SML (Geoffrion 1990) which would provide the declaration dimension of the modeling environment;

(3) a graphical interface for process schema specification similar to that developed by Muhanna and Pick (1988);

(4) support of embedded SQL to facilitate integration with existing data resources.

Implementing MMS with these properties is best undertaken using as many existing tools as possible. An object-oriented language such as C++ would provide the necessary underlying message passing primitives and inheritance structures while also supporting the development of graphical interfaces and embedded SQL retrievals. Perhaps the biggest difficulties would be encountered in implementing demons and the associated dynamic properties of process synchronization. Although some operating systems, such as the Mach version of Unix, provide powerful process

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

communication primitives, it may still be necessary to construct an event-driven calculus similar to that underlying the Simscript language. The development of MMS to fully support model integration requires use of assorted software resources; however, the advance of graphical user interfaces with "clipboard" and dynamic data exchange features should contribute to the feasibility of developing such MMS.

7. Conclusion

Our contention is that a language to support a wide range of model integration contexts must exhibit properties of both math programming and DES languages. It is desirable to enforce separation between model representation and manipulation, as in math programming, so that models can be manipulated in a way that is independent of any particular application. On the other hand, a rich set of manipulation operators, as in DES languages, must be available in order to build complex, multiparadigmatic models such as Figure 2-1.

Given this seemingly disparate set of requirements, what would an integration language with DES features look like and how would it fit in the spectrum of executable modeling languages developed for math programming? The full, structural details of such language are beyond the scope of this paper, but consider briefly one possible scenario in which an existing EML such as SML (Geoffrion 1990) is embedded in a process-oriented language similar to the MICPs shown in §4 and §5. In this context, representations of the model components in Figure 2-1 would be developed independently as structured models using SML. In order to evaluate the integrated model, a MICP would be developed in which the structured models are declared as model object types in the MICP. Each "send" command in the MICP would then activate an evaluation process associated with the specified model object. For example, the command "Send Forecasting.PositPrice" would activate SAS to solve the associated econometric forecasting model (after perhaps converting the SML representation to the appropriate SAS format). In this way, a useful blend of the declarative power of structured modeling and the procedural flexibility of communicating processes would be realized.

In addition to supporting the process-oriented constructs identified in this paper, an integration language implemented in this form could also facilitate the integration of data and models by supporting embedded data manipulation operators—for example, embedded SQL commands if a relational environment were desired. By having models as object types, integration by consolidation would also be possible in that structured model schemas would be available for editing and eventual schema integration as suggested by Geoffrion (1989a, b). It is easy to imagine a MICP with a preamble defining consolidations, and, then the consolidated models are merely treated as components for process integration.

Building a process-oriented language on top of an existing EML is only one possible approach; the converse is to embed process-oriented constructs into an EML, as GAMS does to a certain extent. Other approaches might include adapting object-oriented data and programming environments to provide the functionality we have described. There are a variety of fruitful directions to explore with an eye toward developing a unified model integration language that will allow both structural and behavioral aspects of model integration to be addressed.*

* Robert W. Blanning, Associate Editor. This paper was received on April 3, 1990 and has been with the authors $7\frac{1}{2}$ months for 2 revisions.

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms

Acknowledgements. We wish to thank the Associate Editor and the three reviewers for their helpful comments. Research funding from the U.S. Department of the Army (Grant N62271-86-M-0209) and the Division of Research, School of Business, University of Michigan are gratefully acknowledged.

References

Blanning, R. W., "A Relational Framework for Join Implementation in Model Management Systems," *Decision Support Systems,* 1, 1 (January 1985), 69–81.

———, "An Entity-Relationship Approach to Model Management," *Decision Support Systems,* 2, 1 (March 1986), 65–72.

Bonczek, R. H., N. Ghiaseddin, C. W. Holsapple and A. B. Whinston, "The DSS Development System," *Proceedings of the National Computer Conference,* 55 (June 1986), 442–455.

Bradley, G. H. and R. D. Clemence, Jr., "Model Integration with a Typed Executable Modeling Language," *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Vol.* III, Kona, HI, January 1988, 403–410.

Brooke, A., D. Kendrick and A. Meeraus, *GAMS: A User's Guide,* The Scientific Press, Redwood City, CA, 1988.

Dolk, D. R. and D. J. Kridel, "Toward a Symbiotic Expert System for Econometric Modeling," *Decision Support Systems,* (in press).

Fourer, R., "Modeling Languages Versus Matrix Generators for Linear Programming," *ACM Transactions on Mathematical Software,* 9, 2 (1983), 143–183.

———, D. Gay and B. Kernighan, "A Modeling Language for Mathematical Programming," *Management Science,* 36, 5 (May 1990), 519–554.

Geoffrion, A. M., "An Introduction to Structured Modeling," *Management Science,* (May 1987).

———, "Reusing Structured Models via Model Integration," *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Vol.* III, 1989a, 601–611.

———, "Integrated Modeling Systems," *Computer Science in Economics and Management,* 2, 1 (1989b).

———, "The SML Language for Structured Modeling," Working Paper No. 378, Western Management Science Institute, UCLA, Los Angeles, CA 90024, August 1990.

Hoare, C. A. R., *Communicating Sequential Processes,* Prentice-Hall, Englewood Cliffs, NJ, 1985.

Liang, T. P., "A Graph-Based Approach to Model Management," *Proceedings of the International Conference on Information Systems,* San Diego, December 1986, 136–151.

Miller, L. and N. Katz, "A Model Management System to Support Policy Analysis," *Decision Support Systems,* 2, 1 (March 1986), 55–64.

Muhanna, W. A. and R. A. Pick, "Composite Models in SYMMS," *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Vol.* III, Kona, HI, January 1988, 418–427.

Oren, T. I., B. P. Zeigler and M. S. Elzas, *Simulation and Model-Based Methodologies: An Integrative View,* Nato ASI Series, Springer-Verlag, Berlin, 1984.

Pritsker, A. A. B. and C. D. Pegden, *Introduction to Simulation and SLAM,* Halsted Press, John Wiley & Sons, New York, 1979.

Roy, A., L. S. Lasdon and J. Lordeman, "Extending Planning Languages to Include Optimization Capabilities," *Management Science,* 32,3 (March 1986), 360–373.

Sagie, I., "Computer-Aided Modeling and Planning (CAMP)," *ACM Transactions on Mathematical Software,* 12, 3 (September 1986), 225–248.

Shapiro, J. F., "Decomposition Methods for Mathematical Programming/Economic Equilibrium Energy Planning Models," *TIMS Studies in the Management Sciences.* 10, 1978, 63–76.

SIMSCRIPT II.5 Reference Handbook, (Second Ed.), Jay E. Braun (Ed.), CACI, 12011 San Vicente Boulevard, Los Angeles, CA, 1983.

Sprague, R., "A Framework for Research on Decision Support Systems," *MIS Quarterly,* 4, 4 (1980), 1–26.

Widman, L., K. Loparo and N. Nielsen (Eds.), *Artificial Intelligence, Simulation and Modeling,* John Wiley and Sons, New York, 1989.

Zeigler, B. P., *Multi-Facetted Modeling and Discrete Event Simulation,* Academic Press, London, 1984.

This content downloaded from 205.155.65.56 on Wed, 27 Jul 2022 19:21:51 UTC
All use subject to https://about.jstor.org/terms