A Framework for Highly Reusable Simulation Modeling: Separating Physical, Information, and Control Elements

David B. Pratt, Phillip A. Farrington, Chuda B. Basnet, Hemant C. Bhuskute, Manjunath Kamath, Joe H. Mize

School of Industrial Engineering and Management
Oklahoma State University, Stillwater, Oklahoma 74078

ABSTRACT

This paper presents a conceptual framework for simulation modeling which emphasizes model reusability. Reusability is achieved through the provision of separate modeling constructs for physical, information, and control/decision elements of a system. These constructs contribute to a modular environment in which the modeling elements, once created, can be stored in a modeling database and retrieved. While this framework can be implemented within traditional simulation modeling environments, an object oriented modeling environment appears to offer a potentially superior approach.

INTRODUCTION

One of the most significant criticisms of traditional simulation modeling has been the lack of reusability of the simulation models. This criticism stems from the fact that most simulation models are developed for a specific simulation experiment objective. Once the objective is realized, the model is often discarded. When a new objective is encountered, a new model is created from scratch even though it may include elements contained in earlier models.

This paper presents a conceptual framework for simulation modeling which emphasizes reusability. Reusability is achieved through the provision of separate modeling constructs for physical, information, and control/decision elements of a system. Definitions of these elements are provided in the Appendix. Visualizing physical and information components of a system as distinct elements is straightforward. In most cases, these elements are tangible and easily defined: a lathe or a drill press is a physical component, while a bill of material or a routing sheet is an information component of a manufacturing system. Control components are potentially more difficult to grasp.

When a control element interacts with the physical element it controls, it evaluates the state of the system on the basis of physical system status and other available information. Then an action is taken (i.e. a decision is made) based on an algorithm or a decision process. The decision is then communicated to physical, information or other control/decision components. In this way, control/decision components provide the interconnections between physical and information elements. This interaction is illustrated in Figure 1.

If the issue of reusability is not explicitly considered during the model development phase, the modeler, in an effort to shorten the model development cycle, usually does not make an effort to modularly separate the above components of a model. The elements of information structure, control strategy, or both are hard coded and dispersed throughout the model. The reusability of the model is thus limited. If it is later desired to model a different information flow pattern, the model would have to be almost totally rewritten. Similarly, to model a different control strategy, the model would have to be rewritten due to the embedded code.
Within the proposed framework, however, the task of modifying the model is reduced to replacing the appropriate information and/or control element(s) with a new one(s) from the modeling database, a relatively simple task.

By selecting physical and information components (from a library of model constructs) and interconnecting them with control components, complete system models can be constructed. The advantages of constructing system models under this conceptual framework are: 1) The modeling components are relatively independent of the specific model within which they are used; 2) The components can be designed, tested, and documented as stand-alone units and stored for future use; 3) The components have value outside a specific system, since they can be directly reused; 4) Models are easily modifiable and highly reusable since components can be readily interchanged to modify specific physical, information, or control aspects of the model.

The separation of system elements and consequent reusability is possible because of a major paradigm shift which is currently underway relative to the modeling, analysis, and design of complex operations (e.g., manufacturing systems). Fundamental to this paradigm shift are the recent advancements in object-oriented programming and related technologies. A research team at Oklahoma State University has been exploring alternative approaches to the modeling and simulation of complex systems since 1985. Research to date has resulted in a prototype modeling environment which demonstrates the feasibility of creating logical/decision modularity among the decision elements of a modeled system. This is equivalent to "plug compatible decision elements", which can be replaced as desired in any location in the model. This prototype environment permits the distinct and separate specification of physical elements (machines, material handlers, etc.), decision elements, and data/information elements.

This paper reviews, in greater detail, the issues underlying the separation of physical, information, and control objects in constructing a simulation model. The next section stresses the importance of separation of system elements. The following section discusses the development of models using this framework within a traditional simulation environment. The next two sections introduce the object oriented paradigm and discuss the development of models using the separation framework within this environment. Finally, concluding remarks are presented. The Appendix provides definitions of terms.

SEPARATION OF MODELING ENTITIES

The performance of a manufacturing system is highly influenced by control policies used in its operation. In evaluating the system performance, it is desirable not only to consider the physical components and the physical configuration, but also the results of different operating policies. A complex hierarchical decision making structure exists in a manufacturing system. The decisions are based on available information, which is often incomplete, inaccurate, and delayed. A decision maker at each level uses heuristics, personal expertise, company rules, and policies to arrive at a control decision. Traditional system modeling tools do not provide convenient structures for specifying these interactions. For example, in simulation modeling the representation of controlling influence is often embedded into those elements of code which represent physical components.

There are two reasons why a modeler should incorporate explicit and separate information processing and decision making structures into his/her models: 1) to facilitate a higher degree of reusability, and 2) to facilitate a more "natural" model building environment. Traditional simulation languages do not provide natural constructs for separately and distinctly modeling physical elements, information flows, and control decisions. In addition, the constructs provided for information and control are frequently hard coded and dispersed into the model. This results in code that is hard to modify and difficult to use for multiple purposes.

Designing for reusability involves identification of behaviors that are useful in more than one context. In general this implies a system design which adheres rather strictly to the "one component - one function" doctrine. If a component performs more than one function (i.e. physical, information, and/or control/decision), its usage becomes limited to situations in which all of its functions are required. On the other hand if a strict one-to-one functionality is maintained between component and function, then the components truly become "building blocks" from which a total system model can be constructed. This quite naturally leads to the question of how to separate functionality within a manufacturing context.

Let us begin by presenting more complete definitions of the three components that describe a system; physical, information, and control objects. A physical object is an object with a tangible correspondent in the real
world system. An object can be classified as a physical object if the primary focus of the modeler's interest in the object is its physical extent or characteristics. Examples include parts, machines, and material handlers. An information object is an object which may or may not have a tangible correspondent in the real world system. An object can be classified as an information object if the primary focus of the modeler's interest in the object is its information content. Examples include bills of material and routings. A control decision object is a logical object which typically has no tangible correspondent in the real world system. An object can be considered a control object if its primary function is to (1) evaluate the state of a given system, (2) exercise a logic algorithm, and (3) signal an appropriate action to be taken.

In the proposed framework, information flow is modeled in a hierarchical structure [5, 17]. For example, a customer order is translated into product orders, which, in turn, give rise to component orders, batch orders, and finally, device activity orders. The information processing, and information delays manifest themselves in the creation of these orders. Similarly, the decision making processes are modeled through a hierarchy of control levels: system level control, shop level control, work center level control, and finally, device level control. These control levels can be modeled by expert systems.

Another advantage of the separation of physical, information, and control objects is that it allows the system modeler to think of these elements independently during model development. This provides a more "natural" modeling environment. In other words, when developing the physical model, he/she need not be concerned with information or control aspects. The process involves selecting the appropriate physical components without being constrained by concerns regarding how to model information flow. Similarly, information flow is considered without regard to physical objects. This independence facilitates the creation of more realistic models with a higher degree of integrity and greater flexibility relative to experimentation with the model.

SEPARATION: TRADITIONAL PARADIGMS

The phrase "traditional paradigms" refers to discrete event simulation using procedural languages such as FORTRAN, Pascal, C, or simulation languages such as SIMAN, SLAM II, and GPSS [12, 13, 15]. In these environments, if separation of physical, information, and control elements is not taken into account, simulation logic is dispersed and embedded in the model. This severely limits the reusability of the model. For example, if a system being modeled has alternative routings, the routing decision could be modeled via probabilistic branching in SLAM II simulation language as shown in Figure 2 below.
Under a different scenario, the modeler may have identified the information and physical aspects of the system, enabling him to model the system in a different way as shown in Figure 3. The queue node and the service activity model the physical aspects of the system and the information component is modeled by specifying the attributes of the arriving entities.

In this situation, the modeler is using the information stored in the attributes of the incoming entities as a basis for alternative routing decisions. Since the attributes are set at the upstream portions of the model, the modeler has more flexibility over alternate routing decisions. The modeler can model several different alternate routing decisions by simply assigning the appropriate values to the attributes. This type of modeling requires the modeler to clearly understand why a particular input entity is routed to one of the two alternative routings.

```
Atrib[1] = 1
```

Batch Arrival ... NQS  Q1

```
Atrib[1] = 2
```

![Network Model Illustrating Separation of Information and Decision Components](image)

If the modeler also identifies the control components of the system, he/she may model the system as shown in Figure 4. In this approach, the modeler has clearly identified the NQS select node as a control component of the system. This control component adds more flexibility to the model. NQS is a user written FORTRAN function in which the user has access to all the system parameters. This NQS function enables the modeler to specify different control policies that have complex decision logic which may more appropriately reflect the decisions taken in the actual system. This type of model facilitates plug compatibility of control/decision policies for each control policy. For example, one of the decisions may involve consideration of queue lengths at the succeeding queues Q1 and Q2, while another may require confirmation of the availability of certain resources before routing input entities to a particular queue. By using a framework of separating the system's physical structure from its information content and control logic, alternative decision policies for selecting alternative routings can be quickly and easily tested and evaluated.

**INTRODUCTION TO OOP/OOM**

The concepts of object-oriented programming (OOP) are having a profound impact on computer software construction. Advantages of OOP over traditional (procedural) programming have been documented in Cox [3] and Meyer [8]. According to Meyer [8]: "...object-oriented design may be defined as a technique which, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs." Meyer [8] goes on to point out that objects remain more or less stable, whereas functions tend to adapt to changing needs or circumstances. In terms of simulation modeling requirements, following the object-oriented paradigm has the important advantage of preserving the bulk of developed code for general use in model building. Each model building exercise then performs the particular functions that are of interest at that time. The object definitions remain independent of the functions of the system being modeled. The characteristics of OOP allow us to rethink our entire approach to systems modeling using computers.

Many of OOP's characteristics can be traced to the SIMULA 1 language [9]. SIMULA has found a popular academic following in Europe and throughout the world, but has never gained widespread use in the commercial environment [7, page 105]. While SIMULA embodies some of the concepts of OOP, it is not a pure OOP language. Smalltalk [4], one of the purest OOP languages, was influenced by SIMULA's model of computation. Smalltalk added the message passing paradigm creating a programming style which we now know as OOP [7, page 194; 9, page 437].

The principal idea associated with OOP is that all items (e.g., variables) in the system are treated as "objects". An object is either a "class" or an instance of a class. A class is that software module which provides a complete definition of the capabilities of members of the class. These capabilities are either provided by the procedures
and data storage contained within the immediate class
definition or inherited from other class definitions to
which this class is related. An instance of a class is a
specific realization of the class having all of the
operating capabilities provided in the class definition.
OOP embodies four key concepts which result in
making software systems more understandable,
modifiable, and reusable. These concepts are:
encapsulation, message passing, late binding, and
inheritance.

Encapsulation means that an object's data and
procedures are enclosed within a tight boundary, one
which cannot be penetrated by other objects. Data
stored within an object is directly accessible only by the
procedures that have been defined as part of the class to
which the object belongs. The use of objects therefore
improves the reliability and maintainability of system
code.

Message passing is a necessary result of encapsulation.
In order for one object to affect the internal condition of
another object, the first object must request (by sending
a message) the second object to execute one of the
second object's procedures.

Binding refers to the process in which a procedure and
the data on which it is to operate are related. In contrast
to early binding (i.e., at the time of code construction)
in traditional procedural languages, late binding
provided in OOP delays the binding process until the
software is actually running. This feature provides for
variable data types to change during execution, operator
overloading (i.e., same message structure for different
code implementation), and storage independent of data
type.

Inheritance provides for a low level form of software
reuse. OOP classes are defined in a hierarchical tree
structure. Each class in the tree structure inherits the
methods and data storage structure of all of its
superclasses. Inheritance allows the construction of
new objects from existing objects by extending,
reducing, or otherwise modifying their functionality.

The differences between software development in
procedural languages and OOP languages is due to
these four characteristics. First, understandability of
classes is improved because they represent the data and
method implementations of a coherent concept rather
than the loose combination of multiple procedural
routines. Secondly, the four features of OOP improve
the ease with which already developed software systems
can be maintained and modified. By encapsulating the
data and methods which use the data, internal class
implementations can be altered while instances of the
class retain the same message passing relationships to
other objects in a software system. Finally, base
language code is reusable through inheritance (i.e.,
definition of new subclasses) and through the use of
instances of a class as an internal component of new
classes.

The concepts underlying OOP can be extended to
simulation modeling [1, 6, 10, 14, 16]. The following
section describes recent developments pertaining to the
separation of modeling entities in an object oriented
paradigm. This work is part of an ongoing research
program at Oklahoma State University to develop an
object-oriented modeling (OOM) environment for
manufacturing systems.

SEPARATION: OOM PARADIGM

One approach to implementing an OOM environment
which specifically targets model reusability is through
the development of a modeling object database. One
component of this database is a library of "modeling
primitives". A modeling primitive is a physical,
information, or control/decision object whose
representation portrays the structure and behavior of a
component of the real world system. An object is
considered a primitive if decomposition into simpler
modeling objects is not required for any potential
simulation experiment objective.

As we have seen in traditional simulation languages, the
modeler models the system with elements such as queue
nodes, create nodes, terminate nodes, etc. that are not
directly identifiable in the real system. In an object
oriented paradigm, there is a more direct and natural
mapping between objects in the real system and objects
in the simulated system. The Smalltalk-80 simulation
objects in the object oriented modeling (OOM)
environment developed at Oklahoma State University
can be classified into two broad categories. The first
category contains objects providing the software
functions which allow the background simulation
processing tasks, such as time advance, event
triggering, entity creation, list processing, etc., to be
performed. The second category includes objects
providing the reusable building blocks for modelling
manufacturing systems. It is here that the separation of
physical, information and control elements is utilized
and maintained.
The building blocks for manufacturing modeling consist of general elements found in a manufacturing environment, such as machines, material handling vehicles, conveyors, work orders, routings, etc. As the capabilities of the system are enhanced, the inheritance feature of object oriented programming [OOP] can be utilized to create subclasses of these generic objects that more completely model the behavior of specific items. For example, the modeling of a general conveyor can be extended to such specific conveyor types as a gravity conveyor, or a power and free conveyor, etc. Likewise, the specific types of material handling vehicles can be extended to include manually operated lift trucks, fork lifts, AGV’s, etc. These enhancements are readily achievable in the Smalltalk-80 paradigm because the more specific subclasses automatically inherit the capabilities of their more general parents. The building blocks embody a higher level of abstraction than utilized in currently available simulation languages. For example, the OSU OOM environment utilizes a class of objects called workstations, which are made up of basic modeling primitives: an input queue, an output queue, a processor, and a queue controller, as illustrated in Figure 5 below.

![Figure 5: Primitives Required to Model a Machine](image)

The queue and processor primitives embody the physical aspects of the machine while the controller models the decision making aspects. Thus a machine is a combination of physical and control/decision components that have been combined to model a real world entity. The queue and processor elements embody very limited behaviors. The queue simply holds parts until called for by the processor via the controller. The processor simply delays parts for some prespecified period of time in order to simulate the processing of a part. The controller on the other hand embodies more complex logic. It is essentially the linkage between the input and output queues and the processor. This is illustrated in Figure 6 which shows the dialog between these three primitives.

**Figure 6
Internal Machine Dialog**

Notice in the course of this dialog that the processor (a physical object) does not talk directly to its queues (also physical objects). All conversations occur through the controller which serves as the interconnection between the primitives. Other types of controllers that a machine object might have include tool controllers, jig and fixture controllers, breakdown and repair controllers, and information exchange controllers. In each case, these controllers would be connected on the "other side" to an appropriate physical or information object.

Other classes of controllers are also required. In fact, in a coupled model [see Appendix for definition], a hierarchy of controllers is required. Imagine a coupled model which represents a part of a manufacturing facility composed of departments, some of which are composed of workcenters, some of which are composed of machines. For this hierarchical model, each machine, workcenter, and department would have its own set of controllers.

In general for such a structure, a high level controller cannot communicate directly with a physical object; it must pass messages through the defined hierarchy of lower level controllers. This protocol is required to maintain order. Without it, there would be chaos analogous to the chaos we see occurring when managers circumvent the chain of command.

Interconnections of controllers with physical objects and controllers with other controllers is handled with pointers. Each controller maintains a list of pointers to
the set of objects it is controlling. Note that very much like the physical configuration of a facility, once the pointers are established for primitives and coupled models, they will in general be very static. Since this configuration can be saved within the modeling database, it need never be recreated or modified unless the real world changes. As the real world does change, the corresponding modeling elements can be updated quickly and easily in order to maintain a viable, up to date modeling database.

CONCLUSION

Recent advancements in object-oriented programming, knowledge engineering, engineering workstations and related technologies provide a strategic opportunity for major advancements in modeling methodology. In particular, OOP makes it convenient to implement a framework for highly reusable simulation modeling through the separation of physical, information and control elements of a system. Once generalized system elements are defined, the inheritance feature of OOP permits building of a model database of more specialized system elements. These elements can then be retrieved from the object library and reused as needed.

A prototype modeling environment at Oklahoma State University has demonstrated the feasibility of these concepts. More work is in progress to develop an environment that will bring the concepts into practice [2, 11].

ACKNOWLEDGEMENT

The authors gratefully acknowledge the support provided by grants from the AT&T Foundation and the Oklahoma Center for the Advancement of Science and Technology (OCAST).

REFERENCES


APPENDIX: DEFINITIONS

Control/Decision Object
A logical object which typically has no tangible correspondent in the real world system. An object can be considered a control object if its primary function is to (1) evaluate the state of a given system, (2) exercise a logic algorithm, and (3) signal an appropriate action to be taken.

Coupled Model
A model composed of primitive objects or other coupled models linked by control objects in such a way as to form a coherent entity. A coupled model may be stored as a single entity in the object library.

Discrete Part Manufacturing System
A manufacturing system whose parts are considered as individual independent entities rather than as a continuous stream of homogeneous product.

Information Object
An object which may or may not have a real tangible correspondent in the real world system. An object can be classified as an information object if the primary focus of the modeler's interest in the object is its information content. Examples include bills of materials and routings.

Manufacturing System
A system whose primary function is to add value through the physical transformation and/or reconfiguration of material.

Modeling Primitive
A physical, information, or control/decision object whose representation portrays the structure and behavior of a component of the real world system. An object is considered a primitive if decomposition into simpler modeling objects is not required for any potential simulation experiment objective.

Physical Object
A physical object is an object with a tangible correspondent in the real world system. An object can be classified as a physical object if the primary focus of the modeler's interest in the object is its physical extent or characteristics. Examples include parts, machines, and material handlers.