Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

A perspective on object-oriented simulation

Stephen D. Roberts
Regenstrief Institute

and

Joe Heim
School of Industrial Engineering
Purdue University
West Lafayette, IN 47907, USA

ABSTRACT

The object-oriented view is that a system is composed of interacting objects. These objects are defined so that they hide those data and procedures which implement their behavior, however they can be manipulated by invoking publicly accessible methods defined for their class. These methods are invoked through messages which objects send to one another. New objects can be constructed from others through inheritance and subclasses. Functions and operators can be extended by overloading. Many of these features are available because of dynamic binding provided by object-oriented languages. There are a wide variety of object-oriented languages and some have facilities specifically for simulation. There is little doubt that future simulation languages will incorporate more ideas from an object-oriented perspective, especially as a means of extending the language to a wider variety of applications.

1. INTRODUCTION

If you are engaged in the application of discrete event simulation, you probably simulate the behavior of various objects. In a few instances, the objects may be mathematical or statistical entities, but in most cases, the objects are physical and observable. Perhaps the objects are transporters in a material handling system, tellers serving people in a bank, patients being treated in a hospital, trucks being routed in a distribution system, planes in a flight pattern, disk drives functioning in a computer, switching systems handling traffic in a communications network, etc. Such objects are usually the central focus of the simulation studies and are easy to identify. The simulation problem is generally one of finding a convenient means of modeling the objects and eventually simulating their behavior. By simulation, we will tend to mean discrete event simulation, although many of our observations are independent of the type of simulation. Furthermore we will be most concerned with modeling objects rather than simulating them.

How you approach a problem with simulation is intimately dependent on the modeling tools you have available and your knowledge of their use. Your modeling perspective will be influenced significantly by the tools inherent in the simulation language you use. If you use a general programming language (e.g., FORTRAN) rather than a simulation language, your perspective will be further fixed by the tools you are able to craft from that language you chose (we assume that the simulation will be performed on a computer). Therefore, the questions about the "best" simulation language or simulation platform must be answered in the context of available resources and whether the particular simulation tool meets your needs. How have the simulation tools evolved to meet the variety of simulation needs? How are the tools made available?

1.1. A History Lesson

If we reflect on the three decades of computer-based simulation, we can observe an evolution of simulation languages that generally parallels the development of general purpose programming languages (e.g., FORTRAN, COBOL, Ada). During the 1950s, simulation was exclusively done using general purpose programming languages and sometimes assembly language. Since the computer was used strictly as a device for numerical analysis, the simulation studies were usually numerically oriented. Procedures for generating random variates and collecting statistics were not readily available, so much of the attention in simulation was focused on creating mechanisms for executing simulations. GASP and SIMSCRIPT were among the early simulation languages developed during the 1960's. GASP was an attempt to augment FORTRAN by providing some data structures and functions that were commonly used in executing simulations. SIMSCRIPT went farther by defining a general purpose programming language that contained simulation constructs and procedures. Later in the 1960's, SIMULA was developed as an "extension" to ALGOL to provide simulation facilities to that general purpose language. It should be stated that SIMULA, although never achieving great popularity, also provided some novel constructs for viewing simulations differently (these will be discussed later).

Throughout these efforts to generalize programming languages to include simulation, the emphasis was on the simulation mechanics, i.e. making it easier to produce a working simulation without having to develop your own means of managing a simulation clock, creating random variates, collect statistics, and monitoring simulation progress.

However in the 1960's a different approach to simulation was introduced by GPSS. GPSS was an anomaly because it was the first language to emphasize a specific **modeling** structure and attempted to hide the mechanics of the simulation. Its structure was built upon a predefined class of active entities called "transactions" which flowed through a flowchart of selected operations, much like the logical flow of a computer program follows a programming flowchart. In fact, GPSS was designed so the user could assemble a flowchart of operations and create a simulation without doing any general purpose programming. It was similar to the way that an electrical engineer uses circuit diagrams.

1.2 The Great Language Debates

Throughout the 1970s and even into the 1980s, there has been constant debate in the simulation community regarding the merits of the various approaches. The question was whether a general simulation facility such as SIMSCRIPT was preferable to a modeling facility like GPSS. Certainly

SIMSCRIPT was preferable on the grounds that "anything" could be simulated (programmed?), but GPSS was deemed easier to use and apply. In the late 1970s and early 1980s, we have witnessed a further evolution of simulation languages that blurs their distinction. SIMSCRIPT has acquired more "process oriented" features with the inclusion of resource and process concepts (Russell, 1983) and GPSS has been extended (in GPSS/H, Crain, et al. 1987)) to include many general purpose programming facilities. Also, languages like SLAM (Pritsker, 1986) extended GASP by providing GPSS-like network facilities and including continuous simulation. SIMAN (Pegden, 1982) achieved additional success by also including special facilities for manufacturing simulation such as conveyors and transporters. Newer, as well as the older, simulation languages continue to try to satisfy the dual goals of generality (found in the programming languages) and convenience providing by a set of predefined objects and concepts for direct modeling. The challenge for simulation language designers is to decide exactly what general programming facilities and which predefined objects are needed.

2. OBJECT-ORIENTED SIMULATION

When we describe a situation to be modeled, we define the "things" that are to be modeled. We also declare what each of these "things" can do (valid operations) and what their "state" or condition is before, during, and after each of these operations. So, for a machine shop, we define the machines of interest, the types of parts that are to be produced, and the operations needed to complete their manufacture. The states of the machines are described before a part process, during the process operation, and after completion of a machining function. However, when we begin to create our simulation model we find that the situation which we describe in terms of "things" and operations now must be translated into a different terminology that defines the simulation environment. If we use a network-oriented simulation language, we must somehow transform our problem into simulation entities like transactions, queues, resources, attributes, sources, sinks, etc. It would be more convenient if we could describe our simulation model with the same terminology that we used to describe the actual problem environment. This desire is, in fact, the motivation for special purpose simulations, such as "factory" simulators. Presumably, a factory simulator would contain familiar items like machines and parts and direct methods for production like routings, machining operations, etc.

If we change our terminology just a bit and say "objects" where we have previously said "things," we very easily slide into a recent programming and design attitude change called "object-oriented." Object-oriented programming and object-oriented design is a relatively new approach to software design and development. An object orientation attempts to bridge the gap between the model and what is modeled. The "cognitive leap" that must be made between the physical environment and the computer instantiation is minimized and more faithfully reflects the way the system is being viewed.

Specialized programming languages certainly have their merits and although the language debate continues, it is the other parts of the simulation language, its objects, that make it most attractive to simulation users. Just what objects should a simulation language have? If popularity of the language is any indication, then languages should have certain "stock objects" like transactions, resources, attributes, queues, activities, sources, sinks, statistical distributions, and statistics collectors. If the language is to have applications for manufacturing then perhaps it should have transporters, conveyors, cranes, carriers, and layouts. For other application arenas, other objects may be useful. Describing possible objects could be endless. For instance consider an object for an industrial robot. What properties and behaviors should the object possess -- degrees of movement, mobility, vision, speed and acceleration, grasping orientation, etc.? Can there ever be a completely general robot object defined as simply as a conveyor or transporter? No doubt special cases can be identified and special-purpose modeling facilities constructed. But can they solve the general problem of modeling robots? Human behavior is even more complex and doesn't fit the network model of transactions in a network. As machines and services increase in their ability to behave in a complex fashion, simulation languages which restrict modeling freedom will become increasingly obsolete.

Rather than attempting to provide all possible objects, perhaps the simulation language should have its own facility for defining objects and not depend solely on predefined objects that are furnished with the language. Instead of simulation language vendors adding "features" to a simulation language in response to new and changing user demands, why not allow the user to extend and customize the language by creating new objects and building them on a consistent platform of concepts? This is the goal of object-based simulation and object-oriented simulation languages. Currently, the most prevalent use of object-oriented simulation is based on using an abstract data type as a basis for defining new objects.

2.1 Abstract Data Types

Many proposals for object-oriented designs have their origins in the concept of the *abstract data type* implemented with SIMULA (a simulation language! -- see Birtswistle, et al. 1979). While SIMULA never achieved a large degree of popular success as a simulation language, many of its concepts continue to have considerable influence on programming language design (e.g Ada). For instance, the use of processes and resources are now prevalent in most popular simulation languages. The notion of packages and modules are current constructs in newer programming languages and the abstract data type is the centerpiece of object-oriented programming. An abstract data type is a data type that not only describes its own characteristics, but also defines its own operations. Unlike the familiar "integer" data type and integer operations, which are typically predefined and globally available, an abstract data type can be defined by the user to have a special set of characteristics. Further the operations needed to manipulate the data type can be defined. For example, in a graphics package, a graphical object may be defined to have position, shape, and extent and possess operations that permit it to move and combine. Once declared, the graphical object then becomes a data type within the language. In a data base application, a record becomes a general data type and its operations may provide for storage and retrieval.

2.2 Classes

Typically, a *class* definition is used to define the particular abstract data type. The class definition will specify the implemented data structure and the operations that can be performed. For example, in a queuing simulation, we might define a class for the customer queue as:

```
class customer_queue
    {
    private:
        int current_queue_size;
        int total_customer_encounters;
        float last_customer_arrival;
        float total_waiting_time;
        float max_waiting_time;
        Customer * head, * tail;
        Customer * next ( * Customer );
    public:
        void insert( time t, rank k);
        void remove( time t);
        int size ()
            { return current_queue_size; }
        float average_time_in_queue ()
            { return total_waiting_time /
              total_customer_encounters; }
    }
```

The visibility of the details of the implementation is governed by public and private declarations. The private

portion generally defines the structures of the underlying data type and certain functions that are accessible only by objects within the class. The public portion is accessible by objects outside the class. Obviously, in most common programming languages, data types and their operations are universally visible and the only means of hiding or modularizing is through function or subroutine calls. In the above declaration, the customer_queue contains some private information about its status, its statistics, and a way to find the next Customer. The public access to the queue permits insertions, removals, size query, and summary statistics.

The operations/functions that are defined on a class are sometimes referred to as *methods* or processes. These are similar to functions in other languages. The methods may also be either public or private. Public methods provide the "outside" interface to the class. Objects of different classes deal with each other through the interface methods.

2.3 Formal Objects

An *object* is an instance of a specific class. It contains all the information that a class specifies and actions that may be performed on it by invoking its class methods. The mechanism for invoking methods on an object is by sending it a *message*. For example to remove a customer from the queue, we might write

```
q.remove ( t );
```

which would use the method "remove" (with parameter t) on the object q. The name that describes the type of manipulation is also called the message *selector*. The selector only describes what is desired, not how it should happen.

A message plays the role of a subroutine or function call and will contain parameters that delineate the message. The sending object doesn't need to know how the message will be executed, leaving the details of the implementation to be interpreted by the receiving object.

A system may possess more than one object of a specific class as well as objects of several classes. Because the very existence of the class is defined by the language, object-oriented languages are **extensible** because the user may define new objects with special behaviors. The new objects are defined from the existing classes, objects, and methods available.

2.4 Overloading

One way to extend a behavior is by *overloading*, which permits a method to operate on other objects. For example we may choose to overload the "++" operator for queues so that

```
++q( *Customer );
```

would now add the Customer to the queue.

Function overloading is also permitted. For instance a function which assigns material handlers to pick up a finished part could also be used to assign an AGV. The great advantage of operator and function overloading is that the global number of names needed to describe operations and functions is greatly reduced (you no longer need different names for the same behavior simply because the objects being affected are different). Also as a new object, say a conveyor, is added you can use the same set of material handling functions and simply extend their use to conveyors. Of course, the material handling functions now must identify the objects they are manipulating and choose the appropriate internal functions.

2.5 Inheritance

Most object-oriented systems also permit hierarchies of types through subclasses. A subclass obtains its essential features from its parent class via *inheritance* but can also acquire its own characteristics. Classes and subclasses usually have a convenient message passing system. For example, a convenient inheritance applies when parts are exploded into a production plan and then subsequently assembled. If the various parts are subclasses of the single order, their family of parts can be readily identified and assembled into a common component using a set of assembly instructions specific to the order type. Furthermore, the hierarchies of subclasses permit the formation of sub-assemblies, where components (like the engine within a truck) can be formed prior to a final assembly. Some object-oriented systems permit multiple inheritance in which an object can inherit properties and methods from several different classes. The value of inheritance in programming is that a new class needs only be specified by how it differs from an existing class, rather that being completely redefined.

2.6 Dynamic Binding

Many of the benefits of object-oriented programming result from its *dynamic binding*. When you call a function in a language like C, the compiler and the linker cooperate to generate a call to a physical address. While this is very efficient, you must take care to associate the function with the appropriate data structures. Strongly typed languages attempt to catch mismatched data types at compile time. Others do a poorer job of catching these errors and sometimes the problems are not detected until the output starts looking strange.

In object-oriented programming languages, the programmer is relieved of the burden of calling the right method with the specific data structure. Instead the programmer uses a generic name for the function and the receiving object looks up the proper method. This run-time binding is sometimes called "late binding" and has a number of advantages. In the programming world, late binding means that references are symbolic and methods can be compiled without re-compiling all of its callers. The same symbolic names are used, regardless of the type of object. Finally, a single message can invoke several methods. This is known as *polymorphic behavior* and it allows code to be written that is independent of the receiver.

For simulation, late binding means that the specific machine needed for the part can be determined when the part finishes its prior operation. It doesn't have to be established at the start of the simulation. The time to complete the part can also depend on the current set of resources and the state of the system. A conveyor is handed packages. How those packages occupy space on the conveyor is determined at the time they are placed there, not on some predetermined set of fixed-size bins.

3.0 MODELING PERSPECTIVES

Modeling with objects takes a different perspective. It's not a question of what the objects in my simulation language represent in the real system. It is what are the objects in the real system. If you see your system as composed of general entities, you make them classes. If you need to refer to specific entities, then they should be objects. If these objects appear to collect together, then they can be a sub-class of a more general base class. Methods should be defined for the most general classes. They can be refined for more specific operations if needed. Division into classes, recognition of methods, and the organization of hierarchies form the basic approach to object-oriented modeling.

3.1 Immediate Benefits of the Object-Oriented Approach

A major benefit of object-oriented systems is the design philosophy they bring to a problem. Rather than relying on the processes (procedures) that are found in a system, they focus on the objects. Objects provide both data abstraction and information hiding that help to modularize a problem in its earliest stages of analysis (Tesler, 1986). It stimulates the user

to identify the principal components of a system and to specify their behaviors and interaction. By *encapsulating* the characteristics and methods within the objects, the objects can be viewed as fundamental components of the system, yielding a natural decomposition of a system. The importance of this approach in simulation modeling is that it gives an implementation framework to the common systems analysis and design approach often advocated in simulation studies.

A second major benefit is that simulations become extensible. Existing models can form the basis for new ones and existing concepts can be enhanced to handle new systems. Because objects and their management have a uniform and consistent definition, new objects can be added to existing models. By using function and operator overloading, old symbols take on additional meaning. Inheritance permits new objects to be defined from existing ones by just describing the differences. Old models now become *reusable* because their methods and objects continue to be useful.

Finally, in side-by-side comparisons of object-oriented programming with procedural languages, there has been a substantial reduction in the size of the resulting code (Cox, 1987). The reduced code size means that a single person can manage more complexity. In the simulation of large and complex systems, this benefit can mean that larger and more realistic models are possible without an increase in manpower.

3.2 Potential Long Term Benefits

There are three areas where object-oriented simulations have substantial long term benefits. First, objects in most simulations tend to be physical and real. Generally they can be represented pictorially. Therefore, object-oriented simulation models often have a natural pictorial (iconic) representation and are easily animated. The user can often directly translate his simulation model into an animated simulation without additional conceptual changes. For example, Thomasma and Ulgen(1987) describe the ease of animating the simulation of a manufacturing system in Smalltalk.

Second, because the objects contain their own functionality, intelligence can be built directly into this functionality using the machinery of artificial intelligence and expert systems. For example an AGV can contain within the definition of its class a decision process for choosing among various machines to serve. Further, this "intelligence" may be updated though simulated experience as well as relying on various condition-action rules (Rothenberg, 1986). Eventually objects would be designed that could "explain" simulation behavior as long term information is gathered (Rothenberg, 1988).

Third, objects provide a natural basis for concurrency. The idea would be that each object could be assigned to its own processor and work away until it needed some form of coordination. Although it isn't clear exactly what form the coordination should take, there is a natural division among the simulation components when viewed as objects (Bezivin 1987a, 1987b).

3.3 Some Disadvantages

Several disadvantages have been observed. The run-time cost of dynamic binding is believed to be very high and a topic of considerable interest. Object-oriented environments like Smalltalk require machines with lots of RAM. The world-view of an object-oriented system increases the semantic gap between the language and the actual hardware, which can be a problem for porting simulation models. Also some object-oriented languages require that extensive class libraries be understood before becoming proficient. This increases the learning time and forces users to become more dependent on documentation and high-level debugging tools.

4.0 IMPLEMENTING OBJECT-ORIENTED SIMULATIONS

Many object-oriented systems are attractive for performing simulations. None have clearly been proven to be generally superior as greater research into understanding object-oriented programming continues. Many programming languages now involve objects and what constitutes an object-oriented system is open to debate. Pasco (1986) states that to fully support object-oriented programming a language must exhibit four characteristics: information hiding, data abstraction, dynamic binding, and inheritance. Wegner (1987) argues that an object-oriented language must have objects, classes, and inheritance and that languages with objects are more properly called object-based. Pascal, Algol, or Fortran are neither. Ada, Modula-2, and perhaps C may be called object-based since they support objects as language features. Object-oriented languages would include Simula, Smalltalk (Goldberg and Robson, 1983), and C++ (Stroustrup, 1987; Wiener and Pinson, 1988). It is interesting to note that Simula was designed as a simulation language, Smalltalk contains an extensive set of classes and methods to support simulation, and C++ was designed to be applied to simulation. An example of using Modula-2 as a basis for simulation is found in L'Ecuyer and Giroux (1987) while Samuels and Spiegel (1987) use Ada for their base language. Use of Smalltalk as a simulation language and environment is found in Goldberg and Robson (1983) as well as in the tutorials by Knapp (1986, 1987).

There are several Lisp-based object-oriented languages that distinguish themselves by their lack of strong typing, their abstraction, and dynamic binding. These would include Flavors (which supports a more primitive function than an object), CommonLoops, and New Flavors. A Lisp based object-oriented simulation system is described by Stairmand and Kreutzer (1988). DEVS-Scheme is an implementation of DEVS for hierarchical, modular systems within an object-oriented framework (Kim and Zeigler, 1987). DEVS is a simulation formalism developed by Zeigler(1984).

It should be noted that there are several hybrid object-oriented systems that combine the object-oriented approach with traditional procedural features. For example, Objective-C (Cox, 1987) adds objects, similar to Smalltalk, to the definition of C. So consistent with Smalltalk is Objective-C that a translator has been recently introduced to convert Smalltalk into Objective-C (Cox and Schmucker, 1987). Thus Smalltalk could be used for prototyping and design and later coverted to C through Objective-C for efficient execution. Actor (Duff, 1986) provides a Smalltalk type environment with Pascal-like procedural programming and artificial intelligence features.

5.0 CONCLUSIONS

The object-orientation poses a new approach to simulation modeling and to simulation implementations. For those familiar with GPSS like languages and the process features of SIMSCRIPT, they will see some familiar themes, although couched in a different language. However there is much more. Many of the ideas are based on concepts that were introduced in SIMULA for the express purpose of doing simulations.

Object-oriented systems provide a practical approach for those who design simulation software. Object-oriented languages provide a natural framework for development. The information hiding and abstraction facilities make it easy to develop and maintain complex software components. The extensible platform is an attractive way to add new concepts and features to an existing language. It would appear to be just a matter of time before existing simulation languages attempt to exploit various aspects of object-oriented systems to give simulation modelers access to this powerful perspective.

REFERENCES

Bezivin, J. (1987a). Some Experiments in Object-Oriented Simulation, OOPSLA '87 Conference *Proceedings*, Orlando, FL.

Bezivin, J. (1987b). Timelock: A Concurrent Simulation Technique and its Description in Smalltalk-80, *Proceedings* of the 1987 Winter Simulation Conference, Atlanta, GA.

Birtwistle, G. M., Dahl, O. J., Myhrhaug, B., and Nygaard, K. (1979). *Simula BEGIN*, 2nd ed., Lund:Studentlitteratur.

Cox, B. J. (1987). *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA.

Cox, B. J. and Schmucker, K. J. (1987). Producer: A Tool for Translating Smalltalk-80 to Objective-C, OOPSLA '87 Conference *Proceedings*, Orlando, FL.

Crain, R. C., Brunner, D. T., and Henrikson, J. O. (1987). Advanced Features of GPSS/H, *Proceedings* of the 1987 Winter Simulation Conference, Atlanta, GA.

Duff, C. B. (1986). Designing an Efficient Language, *BYTE*, **11**(8), pp 211-224.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA.

Kim, T. G. and Zeigler, B. P. (1987). The DEVS Formalism: Hierarchical, Modular Systems Specification in an Object Oriented Framework, *Proceedings* of the 1987 Winter Simulation Conference, Atlanta, GA.

Knapp, V. (1986). The Smalltalk Simulation Environment, *Proceedings* of the 1986 Winter Simulation Conference, Washington, DC.

Knapp, V. E. (1987). The Smalltalk Simulation Environment, Part II, *Proceedings* of the 1987 Winter Simulation Conference, Atlanta, GA.

L'Ecuyer, P. and Giroux, N. (1987). A Process-Oriented Simulation Package Based on Modula-2, *Proceedings* of the 1987 Winter Simulation Conference, Atlanta, GA.

Pascoe, G. A. (1986). Elements of Object-Oriented Programming, *BYTE*, **11**(8), pp 139-144.

Pegden, C. D. (1982). *Introdution to SIMAN*, Systems Modeling Corporation.

Pritsker, A. A. B. (1986). *Introduction to Simulation and SLAM*, 3rd Ed., Systems Publishing Corp.

Rothenberg, J. (1986). Object-Oriented Simulation: Where Do We Go from Here?, *Proceedings* of the 1986 Winter Simulation Conference, Washington, DC.

Rothenberg, J. (1988). Knowledge-Based Simulation at RAND, *SIMULETTER*, **19**(2), pp 54-59.

Russell, E. C. (1983). Building Simulation Models with Simscript II.5, C.A.C.I. Los Angeles, CA.

Samuels, M. L. and Spiegel, J. R. (1987). The Flexible ADA Simulation Tool (FAST) and Its Extensions, *Proceedings* of the 1987 Winter Simulation Conference, Atlanta, GA.

Stairmand, M. C. and Kreutzer, W. (1988). POSE: a Process-Oriented Simulation Environment embedded in SCHEME, *Simulation*, **50**(4), pp 143-153.

Stroustrup, B. (1987). *The C++ Programming Language*, Addison-Wesley, Reading, MA.

Tesler, L. (1986). Programming Experiences, *BYTE*, **11**(8), pp 195-206.

Thomasma, T. and Ulgen, O. M. (1987). Modeling of a Manufacturing Cell using a Graphical Simulation System Based on Smalltalk-80, *Proceedings* of the 1987 Winter Simulation Conference, Atlanta, GA.

Wegner, P. (1987). Dimensions of Object-Based Language Design, OOPSLA '87 Conference *Proceedings*, Orlando, FL.

Wiener, R. S. and Pinson, L. J. (1988). *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley, Reading, MA.

Zeigler, B. P. (1984). *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, London and Orlando, FL.

AUTHORS' BIOGRAPHIES

STEPHEN D. ROBERTS is Professor of Industrial Engineering at Purdue University and Professor of Medicine at the Indiana University School of Medicine. His academic and teaching responsibilities are in simulation modeling. His methodological research is in simulation language design and includes INSIGHT, a general purpose, discrete event language, and SLN, for the Simulation of Logical Networks. He is also principal in SysTech, Inc. which distributes the simulation languages and consults on their application.

He received his BSIE, MSIE, and PhD in Industrial Engineering from Purdue University and has held research and faculty positions at the University of Florida. He is active in several professional societies and in addition to making presentations and chairing sessions at conferences, he was Proceedings Editor for WSC '83, Associate Program Chair for WSC '85, and Program Chair for WSC '86. Presently he is Chair of SIGSIM and the TIMS representative to the WSC Board of Directors.

JOSEPH A. HEIM is a Ph.D. student in Industrial Engineering at Purdue University. He received his B.S. in Mechanical Engineering and Master of Engineering in Computer Science degrees from the University of Louisville in 1974 and 1975, and the M.S.I.E. degree from Purdue University in 1987. His research and consulting interests include simulation for design of manufacturing control systems and development of computer supported cooperative work environments.

Stephen D. Roberts
Regenstrief Institute
1001 West 10th Street
Indianapolis, IN 46202, U.S.A.
(317) 630-7447

Joe Heim
School of Industrial Engineering
Purdue University
West Lafayette, IN 47907, U.S.A.
(317) 494-1866