

Simulation Modeling Some Programming Required



IIE Solutions February 1997

by Jerry Banks and Randall R. Gibson

Simulation software advertisements often claim simulation models can be quickly built with "no programming required." It is our view, however, that building a valid simulation model of real systems does require some programming. Programming simply refers to the process of designing, coding, and testing the procedures necessary to imitate real system activities and logic—the real behavior of the system.

This process could take the form of any representation that a computer can interpret and execute, such as an IF-THEN-ELSE type ordered logic written by the user (in a symbolic language), or selected from a list—even using point-and-click operations.

Some simulation software introduced in the last few years attempts to minimize—even eliminate altogether—the programming process. They allow you to select from among pre-built modules on a graphic menu, drag and drop, and click here and there. The result: you're finished! You've built a model, you haven't had to "program." In fact, you haven't had to think too much about it. But be careful; the resulting model may not be accurate enough to be of much use.

Simulation software that requires no programming is very application-specific, relying on pre-built constructs for "typical" activities and functions found in that application. But those constructs tend to be too generic. They can be difficult to modify, and they are not applicable to problems outside of the application domain.

For example, if you are simulating a conveyor, then that conveyor must be of a type supported by the simulation software. Otherwise, there is a risk of approximating your conveyor. Even worse, the result could be major mistakes in the analysis, which would invalidate the results. Obviously, real conveyor systems are complicated and require detailed modeling and careful analysis to correctly simulate their behavior and produce accurate results. Unfortunately, they may be grossly oversimplified in the standard constructs provided with "no programming" simulation software.

Consider the following example. Suppose we use a popular programming simulation software, which is oriented toward manufacturing—call it Software X. The goal is to simulate the loading of a trailer with 10 different rolls of carpet, travel to the destination, then unload the carpet in the reverse order of loading. Using the "LOAD" command accomplishes the placement of the carpet on the trailer. So far, so good. Using the "UNLOAD" command removes the carpet, but in the order that it was loaded. Obviously, the trailer should be unloaded in reverse order. To accomplish the desired result (provided that the modeler knew that the software was not reproducing reality) requires the use of "dummy" locations—fake entities used to allow work-arounds for such problems.

Dummy locations are used frequently in no-programming software to accomplish a desired result, and this can lead to much confusion. Not only is the frequent need to resort to this artifice confusing to the modeler, the use of dummy locations causes other aggravations. For example, in a complex process at a machine, dummy locations may have to be added to portray the various steps in the process. Then, when a report on the utilization of the machine is desired, even more dummy locations have to be added.

Using a different, "programming" simulation software, an order list construct can be used. This allows the modeler to specify to the model that the carpet rolls be unloaded in accordance with the value of an attribute that contains the time of their arrival. Unloading the rolls from lowest to highest time of arrival would accomplish the desired purpose. Although we would consider this a "programming" approach, it seems to us more straightforward and easier to build than the dummy location work around. If the customer later decides that the truck should be unloaded in order of color instead, this too can be easily programmed using the same order list construct.

Generations of Programming A program is a series of instructions that a computer can interpret and execute, allowing it to perform arithmetic and logical operations or comparisons, and perhaps take some additional action based on the comparison, or input or output data in a desired sequence. Programming languages are the syntax, grammar, and symbols or words used to correctly encode the instructions. The first generation of programming languages, called machine languages, required the writing of long strings of binary numbers to represent the basic machine operations, such as add, subtract, and compare. Combining sequences of these instructions to do useful higher level tasks was extremely tedious, and was "prehistoric" programming by today's standards.

Symbolic, or assembly, languages—second-generation languages—were introduced in the early 1950s. They use simple mnemonics such as "A" for add or "M" for multiply, which are translated into machine language by a program called an assembler. Although a big improvement over machine languages, writing programs in these languages was still quite tedious and error prone.

The third generation of languages was developed in the mid-1950s. They were called "high-level" languages because they were largely independent of the hardware (they must be translated into machine code for the target computer by special programs called compilers or interpreters). These were algorithmic or procedural languages designed for solving a particular type of problem. The first such language was FORTRAN (FORmula TRANslation), developed about 1956 and intended for scientific calculations. A number of others followed, many of which are still in use. The most popular are BASIC (Beginners All-purpose Symbolic Instruction Code) and C. These languages allowed much more complex programs to be developed in relatively short order, and allowed the development of many new applications (although still by trained experts). The very first commercially available simulation software (Simula) was a third generation language, and many general purpose simulation software still in use—such as GPSS/H, SLAM II and SIMAN/V—are still third generation procedural languages.

Fourth-generation languages are nonprocedural—they specify what is to be accomplished without describing how. The first one, FORTH, developed in 1970, is used in scientific and industrial control applications. Recent "no programming" simulation software products are based on fourth generation language concepts:

they specify what is to be done, and leave the details of how it is accomplished to the software. However, these products, like most fourth-generation languages, are still written for specific purposes.

Fifth-generation languages, still in their infancy, are an outgrowth of artificial intelligence research. An example is PROLOG (PROgramming LOGic), which is designed for programming logical processes and making deductions automatically. Although there are no fifth-generation simulation products available yet, we expect to see some before the end of the decade.

The User Interface - How Important? The development of user interfaces reflects the evolution of programming languages over the generations, up to the current object-oriented, event-triggered graphical user interfaces (GUI) that we are all familiar with for Windows applications.

An attractive graphical interface that allows the user to easily arrange and connect simplified modeling constructs or blocks is central to no-programming (non-procedural) modeling software. The claim is that menu-based, drag-and-drop, check-the-box type interactions are all that will be required of the user to build, run and test a complete model.

This approach does facilitate the construction of models up to a point. However, there is a point at which a graphical environment presents the user with more barriers than advantages; creating and editing complex models with graphical tools often requires more time than creating and editing such models in a text-based environment. In fact, a no-programming graphical interface can lead the user into a trap: in an attempt to restrict the model building activity to what is supported by the GUI, the user may lose sight of the detail and logic needed in the model while attempting to fit the model to the software capabilities.

User interfaces can be partitioned roughly into three categories:

Interrogative—The software leads the user through the steps. An example is the interview mode in your tax preparation software. The software asks the questions and the user replies.

Declarative—The user fills-in the blanks. Point-and-click software, with some minimal typing, follows this process. Graphical simulation model-building software is an example of this category.

Imperative—The user specifies the actions by some form of programming, even though the programming may be disguised. Traditional general-purpose simulation languages are examples of this category.

Which is the appropriate interface? It depends on the application. For people that have never prepared their own income taxes, the interrogative mode is perfect. After you know what is needed, you can complete your income taxes using the declarative mode, just call up a particular form and fill-in the blanks. The imperative mode is not useful in the case of income taxes since the Internal Revenue Service makes all of the rules and the taxpayer is just the following these. No logic is needed.

The higher the level of the simulation software interface, the less flexibility the modeler is given. The higher the level of the interface, the more decisions are made by the developers of the software, rather than by the user. This trend continues toward higher level declarative, even interrogative interfaces, and we think this leads to overly simplified (often invalid) models.

Why should we be concerned by this trend? The systems that are simulated are usually complex as they have a reactive character. For example, "when the queue in front of drill A reaches or exceeds 20 entities, deploy one of two workers from the sander, provided the sander queue is less than five and there are two workers at the sander, to drill B until the queue at drill A reaches 5 or less." It is impossible or extremely difficult to build an interrogative or declarative interface to describe all algorithmic behavior. Yet capturing this behavior is essential for a valid model.

In some cases models are not so complex. If the subsystem being simulated is isolated from interaction with other subsystems, and there are no conditions in the flow of entities, and if the material handling system is simple, or there is sufficient material handling capacity that it can be treated as a time delay, then an interrogative or declarative interface (no-programming software) can be used for modeling. Such systems are rarely seen in practice, however; in eight years of consulting practice on the part of one of the authors, one such system was seen (out of approximately 100).

The lack of flexibility might be acceptable if there is a way to get underneath the

statements provided by the software. Often, when you get into the language underneath, it is so cumbersome that you wish that you had not even opened the can of worms.

Simulation is a Discipline - Not a Software Advertisements that make the claim of "no-programming required" are written to appeal to the person not trained in simulation modeling. The objective indicated in these ads is that it is possible to hand off the problem to a production line person. The risk is that the person given the task does not have the capability to validate and verify the model, much less understand the intricacies of the model statements, but appreciates the seeming rapidity of the development exercise and the resulting animation and automatic output reports in the form of business graphics if the output looks okay, the person assumes the simulation must be right. Forget about the testing of the model and the statistical analysis—the presentation was made, everyone liked the animation. This approach is dangerous because the model constructed may not reflect the real system that is being modeled. Thus, a Type III error may be committed (i.e., the model solves the wrong problem). We believe that this problem arises, in part, from the "no programming required" misconception. In fact, much of the understanding of the system comes through the difficulties of the model building exercise. The model builder is often forced to resolve conflicts and to explore new directions that were previously not realized or understood. The more automatic the model building exercise, the less this understanding and learning occurs.

Simulation is a discipline, not a software package; it requires detailed formulation of the problem, careful translation or coding of the system logic into the simulation procedural language (regardless of the interface type), and thorough testing of the resulting model and results. There are at least two different skills required to be successful at simulation. The first skill required is the ability to understand a complex system and its interrelationships. The second skill required is the ability to translate this understanding into an appropriate logical representation recognized by the simulation software. Selling simulation software with the lead "no programming required" clouds the need for these two skills.

As simulation consultants, when looking for new staff hires we prefer analysts that have an advanced degree that involves quantitative methods and several simulation and statistical courses. We insist that our new hires have excellent

computing skills, an indication of a logical mind. The discipline could be industrial engineering, operations research, or management science. Many computer science graduates also are trained for a possible career in simulation.

Does Programming = Difficulty? Just what makes building a simulation model difficult? Difficulty is caused by the need for flexibility, and by the need for realistic detail. For example, if a routing decision in a manufacturing model is based on product type or size, or utilization of the possible destinations, or whether or not there is another product of higher priority, and so on, then some logic must be developed to test for these conditions and make the correct logical choice. Hence, difficulty is not a function of model size, number of processes, or the number of different product types. The difficulty is caused by the need to mimic the real-world logic. The appropriate logic is not always so obvious, but it is the heart of most simulation models of real systems. And, it is not always so easy to define this logic with no-programming software. Generally, a programming type modeling environment is more flexible, and supports a wider variety of solution approaches that may be tailored to any degree of detail required.

Having watched the evolution of special purpose simulation products, we have seen software that was easy to use (although not always useful) at its outset become very difficult to use as the vendor became intent on increasing the flexibility so that a process can be modeled accurately, or a material handling device will model the system intended.

For example, the main-building screen of one modeling package has approximately 75 quarter-inch-square icons visible on the screen. Many of these can be clicked on to reveal more icons. It is impossible that the use of such a huge number of icons can be totally intuitive. Eventually, some of these products become so cluttered with drop down menus and hidden options that they are more trouble than programming, just what they were intended to eliminate. This further slows the operation of the software so that succeeding versions go in the reverse direction of what is anticipated, e.g., they become slower on faster computers.

Hidden Dangers The words sound right. You want to model a bridge crane. The no-programming software has a bridge crane! The model must include an

automated guided vehicle. Okay, the no-programming software has an AGV. This looks easy, and the model is constructed rapidly. It takes only four hours to build with animation, and that's four-and-a-half days faster than with the language that doesn't advertise no programming.

But, you had better dig deeper into what the bridge crane module really does. You had better dig deeper into those AGV movement rules. What scheduling logic is being used? What parking rules are used? Is this the same as your bridge crane? Does your AGV system work this way?

There is much danger in treating these capabilities as a black box. You may find that you haven't accounted for the minimum spacing of the AGVs, and that is important. You may find that the control algorithm for the bridge crane is not the same as the one your system is using, and it can't be changed.

We might go so far as to say that by not requiring you to explicitly specify or "program" model logic, no programming packages can and do hide critical detail which you may need to see in order to understand and verify the model's behavior and results. If you were told to design a machine, you wouldn't simply describe the machine's outside view, fill its surface with some pretty colors from your paintbrush type software, and then turn on the automatic inside machine design program. You would need to design the inside of that machine carefully, including all of the spindles, bearings, gears, races, and their interactions.

Here's an example of the "hidden" dangers: A model was developed of a simple high speed packaging conveyer line using the approach promoted by Software Y. The modeler selected from among the high-level blocks provided, clicked and filled in the Windows "check boxes," and accepted some defaults. The modeler made one critical error however, by not specifying a previous conveyer section resource to be released after gaining access to the next section; the modeler inadvertently left the default "release before" option selected. This model construct used an "internal" queue in the queue-seize logic, which is hidden from the user. This queue actually allowed an unlimited number of package entities to build up when the downstream conveyer section filled, providing a totally erroneous result. The model's performance reports provided no clue as to the problem, because the software doesn't report on these hidden queues. Had the modeler used a lower level programming construct instead, this error would have

been much more obvious to detect.

The 80% Syndrome One of the co-authors was enthralled by the possibility of manufacturing-oriented simulation software in the early 80's and became an instructor for a software vendor. A product manager for the vendor—a former student of the co-author—made a statement in confidence, that said in effect: "Even though the ads for this product say that 80% of all manufacturing problems can be solved using this software, I think that it's more like 80% of the problem that can be solved."

We suggest that no-programming simulation software suffers the same limitations: it may get you quickly to the point of modeling about 80% of your system. At that point you begin to discover some shortcomings in the software, and getting additional detail or "fidelity" for the last 20% into your model becomes more difficult. You begin to use building blocks in unusual combinations or use side effects to accomplish your goals. With an inferior package, you may get to only the 85% point and have to decide whether to abandon the software or live with 85% fidelity.

Here's an example that initially looked like a good fit for a "job shop simulator" type of no-programming simulation package. The system to be simulated consists of a small department processing widgets. Widgets were processed at one of four processing machines, followed, by one of two automated inspection machines, followed by joining the widget with a whatzit, then sending the assembly to another department. All processing machines perform the same task, as do all inspection machines. There are two to four operators to perform setups (load machines), takedowns (unload widgets and prepare the machine for the next widget), fix problems and monitor the machines. Once a machine is running (processing a widget) the operator is free to do other tasks. The objective is to determine how many operators are required to keep the machines busy and avoid machine idle time. This problem fits the no-programming simulation software so far. However, there are two additional operational procedures followed that are key to the way the system runs, and therefore must be included in the model. First, at the end of each shift, operators clean each machine and perform a setup—but the machines are not started until shortly after the beginning of the next shift, with a new set of operators. Second, the cleaning takes place usually in the last half hour of the shift if the operator feels there's not enough

time to setup and run another widget; in effect, the operator looks ahead at the clock to make this decision. The existing no-programming simulation software features, such as scheduled PM, can't handle this type of logic—it doesn't fit a "standard" description. To add custom rules would require not only an additional software module (at significant extra cost), but also some programming logic to implement these procedures, even though they are relatively "simple."

If it's necessary to reproduce the real system with a high degree of fidelity, then beginning with software that includes a programming capability is the way to go, allowing the user to go all of the way to 100% fidelity, if necessary.

Conclusion Are we so spoiled by "ease of use" that we can't stand interrogative and imperative interfaces? Not at all! We use them every day (in word processing, spreadsheets, and so on. If fourth generation simulation software with interrogative and imperative interfaces could be constructed to model very complex systems, we would happily endorse them.

What is needed is software that allows operation at multi-levels. We would spend most of our time at the "upper levels" using the interrogative and imperative interfaces (non-procedural definitions), and drop into the procedural, declarative interface only when completely necessary. The problem is the chasm between the interfaces. Thus, we might be drawing networks using point-and-click and shortly thereafter programming in C.

When simulation software can operate smoothly at the three levels, we will be ready to buy it. Until that time, we say that you should take great caution before falling for the notion that complex problems can be solved with no programming.

Acknowledgments The authors appreciate the inputs of Cliff King from F&H Simulations, Alice Cox from Minuteman Software, Matt Rohrer, Paul Stevens and John Carson from AutoSimulations, Incorporated, and Jim Henriksen and Bob Crain from Wolverine Software for their helpful suggestions.

Further Reading

Akbay, K.S. "Using Simulation Optimization to find the Best solution," IIE Solutions, May 1996.

Banks, J., and R. Gibson, "[Getting Started in Simulation Modeling.](#)" IIE Solutions, November 1996.

Banks, J., and V. Norman, "Justifying Simulation in Today's Manufacturing Environment," IIE Solutions, November 1995.

Brewer, S.K., "Simulation: Why Aren't We Where We Should Be?" IIE Solutions, January 1995.

Mabrouk, K.M., "Create Your Own Low-Risk Manufacturing Environment," IIE Solutions, January 1996.

Mazziotti, B.W., "Get More Mileage from Flexible Simulations," IIE Solutions, May 1996.

Porcaro, D., "Simulation Modeling and DOE," IIE Solutions, September 1996.

Routh, G., "Industrial Engineers Play Critical Role in Boeing Celebration," IIE Solutions, July 1995.

Schelasin, R.E.A., and J.L. Mauer, "Creating Flexible Simulation Models," IIE Solutions, May 1995.

Thompson, M., "Simulation-Based Scheduling: Meeting the Semiconductor Wafer Fabrication Challenge," IIE Solutions May 1996.

JERRY BANKS retired from the School of Industrial and Systems Engineering at Georgia Tech in 1999 and joined Brooks Automation, AutoSimulations Division as Senior Simulation Technology Advisor. He is the recipient of the 1999 Distinguished Service Award from INFORMS-CS.

Randall Gibson is president of Diamond Head Associates, a consulting firm based in San Diego, California, that specializes in supply chain strategy, analysis, and performance improvement. He is a senior member of IIE.