Taming The Complexity Dragon

James O. Henriksen
Wolverine Software Corporation
3131 Mount Vernon Avenue
Alexandria, VA 22305-2640

mail@wolverinesoftware.com

Abstract

This paper is an expanded version of a talk I gave at the 2006 Winter Simulation Conference, which is the premier annual conference in the discrete-event simulation community. Each year, the conference recognizes a "titan of the industry," and that person is invited to deliver a 1-hour presentation at a plenary session. I was the titan honored at the 2006 conference. Since this paper was written expressly for members of the simulation community, concepts and terminology peculiar to simulation are not explained. Readers lacking a simulation background may find some portions hard to understand.

The goal of this paper is to present ways in which we can deal with complexity. I believe that the explosive growth of complexity is the computing industry's number one problem. It is the root cause of many other problems, such as computer security. Paradoxically, organizations that are in the best positions to deal with complexity are the ones that are, in fact, creating and expanding complexity.

In the first section of this paper, I discuss the pervasive presence of complexity in our society in general, and in the simulation community in particular. Complexity is of particular importance to the simulation community, because reducing complexity is our primary activity. We analyze complex systems, build models of them, run the models, and draw inferences that yield more easily understood characterizations of system behavior. Reduction of complexity lies at the very core of discrete-event simulation.

In the second section, I develop a framework of software traits, of which complexity is the most important. I use the framework in succeeding sections to discuss the traits and to illuminate relationships among the them.

Following my discussion of the framework, I present four interesting examples of the kinds of complexity I've had to face as a software developer.

Next, having discussed the evils of complexity, I present a sequence of twelve techniques for *reducing*, or at least coping with, complexity. Since model development is a form of software development, we all develop software in one way or another. Some of the techniques are most applicable to software development, while others are more general.

Finally, I present my conclusions.

Complexity is ubiquitous in our society.

> "We're facing a huge problem as a nation. We've made the entire election system overly complex and technologically vulnerable, and lowered public confidence in the legitimacy of the results."
>
> Candace Hoke
> Director of the Center for Election Integrity
> Cleveland State University

In the off-season, the Washington Redskins hired a new offensive coordinator, who brought with him a 700-page playbook. As of this writing, the Redskins are 4-and-9. Late in yesterday's game, the Redskins had the ball on their opponent's 3-yardline when they were penalized for having too many players on the field. Four months into the season, players are so wrapped up in complexity that they can't count to eleven.

Can anyone reading this paper honestly claim the ability to program a VCR without the benefit of a user's manual? Thank goodness the VCR is becoming an obsolete medium.

The computing industry is particularly plagued by complexity. Why is this so? In part it's due to the medium in which we work. When one works with a material such as wood, fundamental limitations are inherent in the wood itself. The world's greatest carpenter cannot build a skyscraper out of wood. While metal and plastic present fewer limitations than wood, the complexity that can be put into a Swiss army knife is bounded. In computing, we work with semiconductors, and on the platforms they provide, we build purely intellectual property. What one can do with silicon and software is virtually unlimited. As a general principle, where there's a lack of constraints, there often is a concomitant lack of *restraint*.

Attacking complexity is difficult:

"I'm sorry this letter is so long. I would have written a shorter one, but I didn't have time."

Some would say that attacking complexity is futile. After all, we live in a complex world, and those who have simple solutions for complex problems fall into one of two extrema. Geniuses are able to cut through complexity and produce simple solutions, e.g., "$E = Mc^2$." Conversely, an idiot's simple solution to a complex problem is likely to reflect his/her failure to understand the problem. There *are* things we can do to combat complexity. We must use all the resources at our disposal to fight the complexity dragon. At the moment, he's winning.

Simulationists Reduce Complexity.

A simulation modeler builds models of complex systems. Modeling is a process of abstraction. A modeler must decide which details of system are important enough to warrant inclusion in a model and which can be omitted. Building a model is the first step in reduction of complexity. A modeler devises a simulation program that executes a model on a computer. The program produces results that in turn must be analyzed to yield insights into system behavior. A modeler almost always uses statistical techniques to analyze program outputs. What is a statistic? A statistic is a function of one or more random variables. It is, therefore, a random variable itself. The mean and variance of a set of random observations are random variables. There's no magic in a statistic. Nothing can make inherent randomness disappear. The utility of a statistic is that it *recasts* randomness in a form that's easier to understand. The mean of a thousand random numbers is easier to understand than the collection of numbers itself. Note that when simulation outputs are insufficiently analyzed, simulation degenerates into being the inverse of statistics. In statistics, we start with mountains of numbers and reduce them to statistical distributions. In simulation, we start with a few distributions and produce mountains of numbers.

We *do* have some techniques for reducing complexity, but we need more, because complexity is growing faster than our ability to deal with it. This paper presents twelve strategies for reducing, or at least coping with, complexity. However, before proceeding, we need to establish a conceptual framework that illuminates the role of complexity. In the following section, I will provide a framework that embodies my characterization of what a software developer needs to do to maximize the utility (read that as sales) of its software.

The Developer's Goal

The developer's goal is to maximize the following measure:

$$\frac{\text{Applicability} \times \text{Correctness} \times \text{Obedience} \times \text{Ease-of-Use} \times \text{Performance}}{\text{Cost} \times \text{Complexity}}$$

For the rest of this paper, I’ll refer to the measure shown above as “our objective function.” Each of the components of our objective function can be assigned a numeric value. Since our discussion of them is purely subjective, we don’t care about their units or scale. Components of the numerator are desirable properties, and those in the denominator are undesirable properties. In the sections that follow, we’ll examine each of the attributes, but we’ll devote the majority of our attention to complexity.

Applicability

Applicability is a measure of whether a software package can be used to solve a particular problem or class of problems. One might be tempted to think of applicability as a 0/1 measure: either a package can be used, or it can’t. In reality, applicability is a continuous measure. For example, we might be able to solve a simple scheduling problem using general-purpose simulation software, rather than using software specially designed for scheduling applications.

The cruelest fate one can suffer is late discovery of inapplicability. You have a project to do, and you choose a software package that seems well-suited for the project. You make rapid progress until the project is 90% complete. Then you begin experiencing difficulties. You encounter important details that either don't quite fit the package's world view or are otherwise poorly supported. As these problems worsen, you find yourself using or combining software features in unusual ways. Eventually, you must decide whether you can live with the level of applicability you can achieve or whether you must abandon the software in favor of a more capable or flexible package. I call this *The 90% Syndrome*.

Later on, we'll talk about software extensibility, which is the ultimate defense against poor applicability. No matter how feature-rich a software package is, it can't do everything. However, if the package can be customized or extended easily, features *not* present can be built from features that *are* built into the software.

Correctness

For the developer, sacrificing correctness makes everything easier, with the important exception of tech support. As a friend of mine once said of optimization, “Optimization is much easier if you're willing to relax the requirement for correctness!” Of course, no responsible software developer is willing to sacrifice correctness. Randomness inherent in typical simulation applications leads simulation practitioners to view correctness in a manner that differs from that of the rest of the world. For us, a 95% confidence interval is a common way of describing the quality of a result. In the real world, if I need to know the balance in my checking account, and my bank's on-line banking software tells me that with probability 95% my balance is between $1,100 and $1,250, I find this totally unacceptable.

The presence of randomness often leads to unexpected behavior in a simulation model. This is precisely why we build models. We hope that any glitches in the operation of a system will appear in a model of the system and not in the system itself. It may be very difficult to distinguish glitches that constitute legitimate (albeit undesirable) system behavior from glitches that are the result of modeling errors, simulation programming errors, or simulation software failures. For precisely this reason, simulation software must be constructed with a zero-tolerance policy for *programming* errors, and it must offer as much support as possible for detecting *modeling* errors. Detection of the former is part of the process of *verification* of a simulation. We need to know if the simulation program doesn’t do what the model says it should do. Detection of the latter is part of the process of model *validation.* We need to know if the model is an invalid representation of the system being modeled.

Poor software architecture alone can be an impediment to correctness. For example, many “older” simulation packages such as GPSS feature fixed pools of entities. The sizes of these pools are determined at the onset of program execution and remain fixed throughout execution. Individual entities are selected by a process of integer indexing. Consider this approach as applied to a simple system having three “A” servers and three “B” servers. A pool of six servers is configured as follows:

Server 1:	A1
Server 2:	A2
Server 3:	A3
Server 4:	B1
Server 5:	B2
Server 6:	B3

Suppose that a program inadvertently references nonexistent model server B4. This error will be trapped, because it is mapped into a reference to the software's server 7, and only six servers are present. On the other hand, consider a reference to model server A4, which is mapped into the software's server 4. This obvious modeling error will not be trapped by the software, and it will result in invalid behavior. This class of undetected errors can be eliminated by providing arrays of servers (discussed later on).

Obedience

With two exceptions, software should do what you tell it to do:

1. Software should disallow obvious errors.
2. Software should complain about questionable uses, but the user should have the final say-so. If I ask the system to reformat my hard drive, I should receive “Are you sure?” and “Are you absolutely sure?” messages. This is called *The Principle of Minimum Regret*. Actions with potentially large negative effects should be hard to perform.

Examples of poor obedience are abundant. Microsoft Word is a good example. How often have you sat down to quickly type a Word document, only to be confronted by messages such as “You appear to be typing a letter. Do you need help?” No I don't. Try typing a numbered paragraph. Word will instantly jump in and assume control of the numbering process, often in ways that you don't like. Try typing the description of a mathematical algorithm using index variables named i, j, and k. Unless you've disabled the pertinent option, Word will convert “i” to “I”. It makes you want to scream “Just leave me alone!”

Ease-of-Use

Suppose you're considering two software packages for use in a given application. You might be tempted to think of the ease-of-use of each package as a numeric value; e.g., Package A might rate an 8, and Package B a 6. For a number of reasons, this “Bo Derek” approach is invalid. Software ease-of-use is not constant; it varies over the life of a project, and local discontinuities are commonplace. As you progress toward completion of a project, you may suddenly encounter something you've not done before with the software. This may necessitate pausing to read the documentation or experimenting with several techniques before continuing. When such events take place, ease-of-use plunges steeply and gradually recovers.

Ease-of-use may exhibit an overall upward or downward trend over the life of a project. For example, software may be advertised as being capable of doing a simple application “in 10 clicks.” For such packages, initial ease-of-use is extremely high; however, as a project progresses, and more complicated details of system operation must be described, ease-of-use will inevitably decline. On the other hand, another package may force you to come to grips with certain key concepts up front. The initial ease-of-use of such a package may be lower than you'd prefer; however, as you progress through a project, you may find that concepts with which you originally struggled, but eventually mastered, are directly applicable, and there's a payoff for your efforts. The ease-of-use of such software will exhibit an overall upward trend. The key to deciding which software to use is determining project duration and complexity. If you have to produce a model in three weeks, choosing the package with declining ease-of-use will be your best option, because the decay in ease-of-use will be experienced over such a short period that it will be insignificant. Conversely, if you're tackling a long, complex project, and especially if you're going to be doing similar projects in the future, choosing the package with increasing ease-of-use is best, because its learning curve will be amortized over a long period of time and many projects.

Software ease-of-use depends heavily on the software's user interface(s). A user interface can be interrogative, declarative, or imperative. The best example of an interrogative user interface is that of tax preparation software, which typically features an "interview" mode, in which users are lead through a step-by-step sequence of questions. In a declarative interface, users specify *what* needs to be done, not *how* it should be done. In an imperative user interface, users provide detailed instructions that specify *exactly* how things should be done. Both developers of and users of general-purpose simulation software must make tradeoffs between declarative and imperative forms of user interfaces. Most users prefer to use declarative interfaces until they reach the point at which paradigms that underlie the interfaces no longer match their needs.

Tradeoffs between ease-of-use and applicability are commonplace. For example, a Swiss army knife may contain a tiny pair of scissors that are nowhere as easy to use as a full-sized pair of scissors. The same is true for many implements in a Swiss army knife; however, if one is going on a camping trip, carrying a Swiss army knife is preferable to carrying a hardware store's worth of gadgets.

Performance

Virtually all simulation models are CPU-bound applications, so performance is extremely important. I have customers who from time-to-time conduct massive 24-hour tests. If my competitors' software runs 10 times slower than mine, I win and they lose, because ten days is too long. Extremely poor performance limits software applicability.

Simulation software performance is the end result of a few big things, a considerable number of medium-sized things, and many, many small things. Examples of large things include a proper event list algorithm (Fishman 00), the use of compiled code vs. interpretive execution (Henriksen 75), and meticulous attention to detail in critical path algorithms. I once had a conversation with a talented developer who had built simulation software in Java. He was pleased that his software could execute 45,000 context switches per second. A discrete event simulation achieves the illusion of true parallelism by sequentially switching back and forth rapidly among activities that take place in parallel, so rapid execution of context switches is critical to performance. My friend asked me how many context switches Wolverine Software's SLX software could perform per second. I thought for a few seconds and responded "45 million." We didn't compare the clock speeds of our respective computers, so SLX might be faster by a factor of "only" 300-500. Context switches are a big thing.

Let's consider a medium-sized thing, taken from the world of animation. One of the tasks that is commonplace in animations of manufacturing and traffic systems is animating the motion of objects along circular arcs. To achieve smooth animation, software must update the positions of such objects each time the screen is refreshed. To keep the explanation that follows simple, I'll confine my discussion to 2D animation. (Arc motion in three dimensions is quite a bit trickier, although the same principles apply.) The natural way of representing arc motion is to use polar coordinates and to transform them into Cartesian coordinates. Updating an object's position along an arc of radius r located at (x_c, y_c) takes the following form:

$$\theta = \theta + \Delta\theta$$
$$x = x_c + r * \cos(\theta)$$
$$y = y_c + r * \sin(\theta)$$

Assume that 1,000 objects are to be moved per screen update. If the screen is refreshed at a 60 Hz. rate, 16 milliseconds of time are available for updating 1,000 objects. Hence, an update for a single object must be completed in 16 microseconds. A microsecond is, then, a very large amount of time. Delving into standard run-time library implementations of sine and cosine functions is very illuminating. One typically finds considerable overhead. For example, PC (X86) implementations of sine and cosine functions restrict the domain of arguments of these functions to magnitudes less than 2^{64}, so library functions may contain explicit tests for rarely encountered domain violations. Explicit tests are unnecessary for two reasons. First, X86 hardware sets an easily interrogated status bit that can be tested to detect domain errors after computing a sine or cosine. Second, underlying software architecture is almost certainly such that the absence of domain errors can be guaranteed. If so, there's no need to test for domain violations. Implementing one's own sine and cosine functions in assembly language greatly reduces overhead.

X86 hardware includes a SINCOS instruction that can simultaneously compute a sine and cosine. (This is an artifact of the cordic algorithm used to compute these functions.) Using SINCOS cuts the cost of an update by nearly a factor of two. Notwithstanding the considerable improvements attainable through careful use of assembly language, directly computing sines and cosines is still expensive. By carefully reconsidering the problem, we can achieve even greater improvements. At each step beyond the initiation of arc motion, we're calculating an (x,y) position that is a function of $\theta + \Delta\theta$. The following trigonometric identities that we learned in school, and we thought to be useless at the time, turn out to be extremely useful:

$$\sin(\theta + \Delta\theta) = \sin(\theta) * \cos(\Delta\theta) + \cos(\theta) * \sin(\Delta\theta)$$
$$\cos(\theta + \Delta\theta) = \cos(\theta) * \cos(\Delta\theta) - \sin(\theta) * \sin(\Delta\theta)$$

For constant rotational speed, the terms including $\Delta\theta$ are constant. After computing the sine and cosine of the initial angle, subsequent updates can be expressed as follows:

$$\sin_{i+1} = \sin_i * K_{\cos} + \cos_i * K_{\sin}$$
$$\cos_{i+1} = \cos_i * K_{\cos} - \sin_i * K_{\sin}$$

Using the formulation above, sines and cosines can be computed using a linear recurrence relationship; i.e., only multiplications, additions, and subtractions are required. Further optimization can be achieved by exploiting multimedia hardware instructions that can perform four arithmetic operations simultaneously. The grand total savings over naive use of library functions is a factor of twenty.

Good performance is the result of many small things. Let's consider an example taken from compiler code generation. The essence of the compiler writer's job is to convert a source program into executable machine instructions. Consider the following sequence of three instructions:

Instruction A
Instruction B (Depends on the result of Instruction A)
Instruction C (Independent of Instructions A and B)

Clearly, there are three valid execution orders for these instructions: A-B-C, A-C-B, and C-A-B. The only restriction is that Instruction A must precede Instruction B. In early CPUs, all three orders were equivalent, because CPUs plodded (by modern standards) through sequences of instructions one at a time. The advent of pipelined CPUs provided faster CPUs, but required more complicated programming to maximize their performance. The basic idea behind a pipelined architecture is partitioning a long operation into a sequence of shorter steps that can be executed in parallel. For example, an instruction that takes 5 machine clock cycles to complete might be partitioned into 5 steps. Given a sequence of repeated instructions, it takes 5 cycles for the first result to make it through the pipeline, but subsequent results become available at a rate of one per clock cycle, assuming independence of the repeated instructions. For early pipelined architectures, the sequence A-C-B yielded the best performance. Why? In the sequences A-B-C and C-A-B, the dependency of instruction B on instruction A caused a bubble in the pipeline, because instruction B couldn't start until instruction A finished. Inserting independent instruction C between instructions A and B accomplished useful work in what would otherwise have been a lost cycle. Present day CPUs have progressed further. Among other things, they can tolerate a degree of instruction dependency by executing instructions out-of-order (o-o-o). Thus, sequence A-B-C is executed as A-C-B, because the CPU automatically recognizes the dependency of B on A, the independence of C, and possibility for o-o-o execution. Since CPUs have finite queues, there exist upper bounds on o-o-o execution. The result is that today, compiler writers work to reduce what is described as “dependency *pressure*,” rather than working to eliminate individual, absolute dependencies.

The code generated by a simulation language compiler contains certain patterns of machine instructions that are generated many times and executed a great number of times. By paying careful attention to dependency pressure, a compiler writer can increase the execution efficiency of compiled code.

Cost

We now move on to cost, one of two terms in the denominator of our objective function. Cost is a squishy concept. Many prospective purchasers of software know what they want, but don't have the money to buy it. For them, costs that others would deem reasonable are effectively infinite, zeroing our objective function. Others have lots of money but don't know what they want. (This is the "Send me a proposal" syndrome.) To such people, cost is relatively unimportant. Academics rarely have *any* money; hence most software companies offer preferential pricing for academics. Paradoxically, the products produced by academics, namely their writing, often *do* have cost. I do a fair amount of research on the Internet. It seems that every time I home in on exactly the paper or article I'm looking for, I get a message that says "Oh, you're not a member of our 'club.' It'll cost you $20 to get this item." I find the asymmetry intriguing.

Cost is most important because of its relationship to the other terms in our objective function. All of the other traits in our objective function have costs. If costs become too high, they become effectively infinite, both for the developer and the user. Thus cost always imposes limits in how far we can go in our attempts to maximize other terms in the numerator of our objective function. Furthermore, cost is proportional to the complexity with which a developer must contend.

Complexity

Exponential growth of complexity is the computing industry's most important problem. It is the root cause of many other problems, such as poor security. While complexity is bad in its own right, complexity also adversely affects other terms in our objective function. Clearly, complexity faced by a software developer adversely affects correctness, ease-of-use, performance, and cost. In extreme cases, complexity affects software applicability. If a product is too complex, we may not be able to use it. Complexity can even affect software obedience. Upon return from the 2006 Winter Simulation Conference, I attempted to export email I had collected using Microsoft's Outlook Express on my laptop computer. My attempt elicited the following message:

> The export could not be performed. An error occurred while initializing MAPI.

The simple, straightforward task of exporting messages turned out to require 45 minute's worth of Google searching and reading. I would give Microsoft a grade of zero for the obedience of Outlook Express.

Complexity I've Faced in Software Development

In this section, I will present four examples of complexity I have faced as a developer. These examples deal with hardware and software issues that most computer users never see. Mark Twain once said "People who like sausage and respect the law should not see how either is made." I hope that by explaining the following examples in layman's terms, I'll avoid having you think that Twain's assertion should also apply to software development.

The most obvious role of a compiler writer is generating machine instructions from a user's source code. The degree of difficulty of performing this task depends heavily on how a target machine's instructions are encoded. The X86 architecture is exceptionally difficult to encode. Among the difficulties imposed are the following:

- Instructions contain packed bit fields that are hard to decode when looking at memory images. One of the first tools I build when constructing a compiler for a new instruction set is a disassembler. For the X86 architecture, having a disassembler that can display generated machine instructions is absolutely essential.
- Instructions can have 1-, 2-, and even 3-byte operation codes (opcodes). Opcodes can be modified by instruction prefixes.
- In some instructions, one 3-bit instruction field specifies a machine register, while in other instructions, the same field is used to extend the instructions' opcodes.
- For some forms of instruction operands, six of the eight possible values of one of the 3-bit fields directly specify a register. One consequence of this encoding is that use of two of the eight available registers requires a more complex instruction format. The two special cases are as follows: one bit pattern specifies that a so-called SIB byte follows, and another specifies that the address of the data to be manipulated by the

instruction is embedded in the instruction as an absolute memory address. An SIB byte is used to specify an index register and a scale factor to be applied to it. SIB addressing is useful for array references. Accessing the i'th element of an array of 4-byte integers can be accomplished by loading the value of i into an index register and applying a scale factor of 4

- Some instructions can be encoded in more than one form.
- Branch instructions have short and long forms. If the destination of a branch is unknown at the time an instruction is generated, the long form must be assumed (in the absence of post-code-generation compaction.)

I recently spent around six months porting Wolverine's SLX software to Microsoft's 64-bit Windows. Doing so was important to our long-term plans, but it was very costly, due to *additional* complexity introduced when the X86 architecture was expanded to 64 bits. In the next four sections, I'll present two examples of increased hardware complexity and two examples of increased software complexity.

Hardware Complexity Example #1

When the X86 architecture was expended to 64 bits, hardware designers rightfully observed that the X86 architecture, with eight general-purpose registers, was register-starved, and they concluded that providing sixteen general-purpose registers would alleviate this problem. Since it was cheaper to use common instruction decoding hardware for use when operating in 64-bit mode and 32-bit “legacy mode,” designers decided to keep the format of 64-bit instructions as close as possible to their 32-bit equivalents. This gave rise to the unfortunate problem of how to encode 16 registers in a 3-bit field. The solution was to provide an instruction *prefix* containing the 4th bits of the various instruction register fields. A total of sixteen prefix values were required. There was no 16-instruction gap in the X86 opcode set, so to make room for the prefixes, designers eliminated eight forms each of the INC (increment) and DEC (decrement) instructions. Unfortunately for me, the SLX compiler generated such instructions. I had to convert to an alternate form of INC and DEC instruction encoding.

Correcting the problems described above was an onerous chore, however, the most serious consequence was that the inherently serial process of generating machine instruction components was disrupted. When generating X86 instructions, it's possible for the most part to generate an opcode and then sequentially generate instruction components that follow. To generate a 64-bit instruction, a compiler must examine all instruction components to determine which, if any, require the use of an instruction prefix. The prefix must be generated prior to generation of the instruction's opcode. This was a foundation-shaking paradigm shift.

Hardware Complexity Example #2

The second hardware-imposed increase in complexity was that designers replaced absolute memory addressing (in which the absolute address of a datum is directly encoded in an instruction) with so-called PC-relative addressing. “PC” is short for “program counter.” When PC-relative addressing is used to access a datum, the address of the datum is formed by adding an *offset* encoded in an instruction to the address of the next instruction to be executed. PC-relative addressing imposes two challenges, one relatively straightforward, and the other devilishly complex. The straightforward problem is that one must know the length of the current instruction in order to determine the location of the next instruction. While this may sound simple, instruction lengths are highly variable, depending on which of myriad formats are used, so properly determining instruction length requires meticulous care. The second problem stems from the fact that data addresses are used in two ways. The most common use is to specify the source or destination of data. However, data addresses are also used as in-line constants, e.g., “Are contents of register X equal to the following data address?” In a 64-bit instruction, data access requires PC-relative addressing, while address comparison is done by encoding an absolute address as an in-line instruction component. In 32-bit mode, there is no such distinction. Addresses are simply absolute memory addresses, and whether they’re used to access data or as in-line constants makes no difference. Porting a 32-bit compiler to 64-bit mode requires identifying all points in the compiler where such distinctions are important in 64-bit mode. For me, this problem was one of staggering proportions.

Software Complexity Example #1

As difficult as some of the hardware issues described above were, they paled in comparison to software issues. The first example I'll give is the 64-bit standard calling sequence. In the 32-bit world, a variety of calling sequences exist for calling functions. In some calling sequences, function arguments are passed by pushing them onto the run-time stack. In the so-called fastcall calling sequence, the first several arguments are passed in machine registers. Depending on whose compiler you're using, fastcall may use different registers. Finally, there are different standards for who restores the run-time stack, the caller or the called function.

To remedy this situation, designers of the 64-bit versions of Windows defined a single calling sequence that is used *everywhere*. While it greatly simplified some aspects of compiler and run-time support development, the new calling sequence introduced new problems. The new standard is a variant of the old fastcall calling sequence. The first four arguments to a function are always passed in registers. The run-time stack must be maintained in the following format:

Local (temporary) Variables
Arguments 5…N
Hole 4
Hole 3
Hole 2
Hole 1
Caller's Return Address

A 16-byte-aligned “hole” for four 64-bit values must be maintained at all times. The alignment requirement stems from the fact that some 64-bit instructions require 16-byte data alignment. The hole is reserved by Microsoft for use by its compilers. The requirements for stack alignment and hole omnipresence are extremely onerous for the compiler writer. Two approaches are available for maintaining the presence of a hole. First, one could open a hole prior to every function call and close it upon return from the function. For consecutive function calls in a non-optimizing compiler, this would lead to “close hole, open hole” sequences, which would be inefficient. Second, a compiler could determine the largest amount of stack space (including the hole) required in a given function, and allocate that much stack space upon function entry. This is the Microsoft-recommended approach.

The 64-bit standard calling sequence is a significant departure from traditional stack-based architecture, in which data is pushed and popped with reckless abandon. 32-bit SLX utilized many “lightweight” calls of the following form:

```
PUSH        save some info
CALL        a function
POP         restore info
```

Unfortunately, all such calls PUSHed data into the hole required by the 64-bit standard calling sequences. Since Microsoft reserves the hole for its sole and exclusive use, in 64-bit mode information saved on the stack in the manner shown above was susceptible to being overwritten. In SLX, such calls were not only present in assembly language run-time support functions; they were generated by the SLX compiler. In 64-bit mode, such calls are still possible in circumstances under which no Microsoft compiled code will be invoked, but the penalty cost for failing to identify the potential existence of such calls is very high.

Software Complexity Example #2

A second example of increased software complexity is the 64-bit implementation of exception handling, e.g., division by zero. General-purpose simulation software simply *must* handle all run-time exceptions. Abrupt, silent termination of a long simulation run due to an unhandled exception is unacceptable. In C/C++, exceptions are traditionally handled by using try/except clauses:

```
Function A
    __try      (Attempt to execute the following code)
               {
               …
               }

    __except ("if an exception occurred")
               {
               Handle the exception.
               }
```

Prior to the 64-bit versions of Windows, the implementation of try/except required execution of a handful of instructions. Exception-handling information was pushed onto / popped from the stack, so if function A (shown above) called function B (not shown), function B could assume control of exception handling (by means of its own try/except logic) and relinquish control back to function A upon exit. There were two deficiencies in the implementation. First, it imposed execution overhead. Second, try/except could be exploited by hackers. By surreptitiously specifying a malicious exception handler and injecting a single invalid opcode into a program, a hacker could acquire control of execution in the middle of an important application, e.g., your email software.

The 64-bit version of try/except is table-based. The tables are horrifically complicated. For each function, the table contains the starting and ending addresses of the machine instructions comprising the function. That's the easy part. A function comprises three pieces: a prolog, a body, and an epilog. If an exception occurs, and system-provided error recovery is to be used, run-time support for exception handling must be able to "unwind" the effects of a function's prolog or epilog. This entails constructing table entries that describe the semantics of individual instructions comprising each. Not surprisingly, this is extremely difficult.

There are three things wrong with this approach.

- It's exceptionally difficult for the compiler writer.
- It cannot properly handle bad branches to locations outside the regions described by the tables.
- Well-designed applications do not alter exception handling conditions frequently. In the case of SLX, execution of an entire program is covered by a single try/except, so any overhead is miniscule.

My solution for handling exceptions in SLX is somewhere between ingenious and downright devious:

- Microsoft provides a Windows API call that allows a program to say "append the following information to the exception-handling tables." Microsoft needs this for just-in-time (JIT) compiled applications.
- Dynamically-supplied table extensions are examined before static table entries.
- I tell Windows that I have a function that is loaded at address zero and extends to the highest address in virtual memory. Since my table entry is examined first, and it covers exceptions occurring anywhere in virtual memory, SLX is always the first to "see" an exception. If SLX's internal exception handler is enabled, it handles the exception with its own custom unwinding. If internal exception handling is turned off, Windows is allowed to process the exception exactly as it would for any application that has no exception handler.

Implementing 64-bit exception handling cost me four days' effort.

How Can We Reduce or at Least Cope with Complexity?

In the sections that follow, I will present twelve approaches for reducing complexity. Any resemblance to the 12-step program for dealing with addictive substances is purely intentional.

Complexity Reduction 1 – Heed the Law of Least Astonishment and the Principle of Minimum Regret

The Law of Least Astonishment states that things should work the way we expect them to work. Expectations of how software should work are formed well before we start using a software package. These expectations come from experience and common sense. After using a good software package for some time, we understand the flavor of how it works, and our expectations are further refined so that when we have to use a feature that's in an area of the software we haven't used before, we have well-defined expectations. If the software behaves differently from our expectations, we have every right to feel disappointed. Violations of the Law of Least Astonishment are abundant. How do we turn off our PC? We press the Start button.

A few days after I began using the X64 version of Windows XP, I noticed an obvious spelling error in a message I was sending using Microsoft's Outlook Express. I thought that I had inadvertently turned off the spelling checker. I turned it on, and strange messages ensued. Seeking help, I turned to the first line of support, Google. I found many articles that told me that Outlook Express didn't have its own dictionary, but that it could use dictionaries contained in other products. A lengthy search for file names and locations yielded no results. Ultimately, I located a Microsoft document that described "implementation differences" between the 32- and 64-bit versions of Outlook Express. One of the differences was that the spelling checker was not implemented in the 64-bit version of Outlook Express. This gross violation of the Law of Least Astonishment cost me 45 minutes.

Let's turn our attention to an example from the world of simulation. In the example that follows, the "advance" verb is used to schedule a time delay in a simulation program.

```
advance    1.0e20   (a giant delay)
advance    1.0      (a small delay)
```

What is the value of the simulation clock after the second delay is executed? The answer lies in the operation of floating point hardware. Double precision arithmetic affords roughly 16 decimal digits of precision. The sum of 1.0e20 + 1.0 requires 21 decimal digits. Low order bits are truncated, so the simulator clock reads 1.0e20 both before and after the second time delay. While this property of floating point hardware is well-known, it's something most people never think about. If simulation software allows such operations to happen without issuing warnings, the Law of Least Astonishment is violated. The intent of the second advance statement is clear; it just doesn't happen. Wolverine's GPSS/H and SLX products have detected and diagnosed such problems for many years. About once every ten years, I hear from someone who's experienced this problem.

We discussed the Principle of Minimum Regret above. To briefly reiterate, this principle states that actions with potentially large negative consequences should be hard to do; e.g., requests to format a hard drive should be honored only after issuance of very stern warnings.

Complexity Reduction 2 – Know when to go back to the drawing board.

X86 hardware provides so-called protection rings for use by operating systems. The basic idea is that lower-numbered rings have greater privileges than higher-numbered rings, and transitions from ring-to-ring are handled very carefully. Ring zero is reserved for the most privileged parts of the operating system. Windows uses only two of the four available protection rings. Device drivers run in ring zero. Since most device drivers are supplied by organizations other than Microsoft, we find ourselves in a situation where millions of lines of non-Microsoft code run in ring zero.

A friend of mine who retired from Hewlett Packard several years ago is a principal in a small startup company named Secure64 Software Corporation. Secure64 produces a secure operating system that runs on Intel Itanium processors. Itanium chips have hardware features that facilitate the use of many rings of protection. Secure64's goal is to have no more than 16K lines of code that run in the equivalent of ring zero in their operating system. That's what I call going back to the drawing board.

Turning our attention back to simulation, I'd like to discuss architectural changes I implemented in Wolverine's SLX software. SLX is in many respects the result of going back to the drawing board after years of experience developing its predecessor, Wolverine's GPSS/H.

In discrete event simulations, model components undergo exactly four types of delays:

- Time-based delays are *scheduled* delays, e.g., interarrival times and service times in queueing systems. Frequently such delays are scheduled by sampling from statistical distributions.
- State-based delays require a specified collection of *state variables* to attain specified values; e.g., “wait until the length of the queue is greater than 10.”
- Hybrid delays are a combination of the previous two types of delay; e.g., “wait until this time is reached or an exceptional condition arises.”
- User-managed delays are typically used when a model has complicated queue management that is not easily described using the previous three types of delay.

In GPSS/H, there are 7-8 different statements for specifying delays. In SLX, there are exactly two statements, “advance” (for scheduled delays) and “wait until” (for the first three types of delay described above). Note that “advance” is provided only for convenience, because scheduled time delays are ubiquitous. “advance” can be regarded as a convenient special form of “wait until.” All forms of delay in a discrete event simulation can be built from these two primitive operations. Implementation of wait until is very complex, requiring sophisticated compiler technology and run-time support; however, this complexity is hidden from SLX users, and using wait until is very easy.

SLX's wait until operates on arbitrarily complex Boolean combinations of state variables. When a model must wait for a complicated condition to be satisfied, queues are constructed for only those state variables that lead to the falseness of the wait until condition. When a state variable is changed, and it has a non-empty wait until queue, model components in the queue are allowed to retry their wait until conditions. In other software, the equivalent of wait until, if it exists, is often implemented by polling (periodic retesting). (This is “Can you hear me now?” mode, and it's very inefficient.)

Going back to the drawing board resulted in reducing the complexity of model delays to two efficiently implemented generic forms that provide a foundation upon which a wide variety of modeling world-views can be constructed.

Complexity Reduction 3 – Do the little things.

As with software performance, reducing complexity requires attention to a small number of large issues and a large number of small issues. One example of a small issue is how best to show the accuracy of a statistical result. One commonly used approach is to build a *confidence interval* for the mean of a statistic. The problem with this approach is that many simulation users/customers don't know what a confidence interval *is*. Others think they know, but their understanding is incorrect. The most common misconception is as follows. “A confidence interval is a pair of numbers and a probability that the correct answer lies between the two numbers.” This characterization makes it sound as though the endpoints of a confidence interval are fixed constants, and the mean value of the system property being observed is random. In reality, the mean value of the system property is fixed; we just don't know what it is. The endpoints of a confidence interval are random variables. If we conduct many experiments using the same procedure for constructing a confidence interval, the probability that any given randomly generated interval will contain the mean is the probability we have defined for the confidence interval.

This summer, I became intrigued with several publications authored by Bruce Schmeiser (Schmeiser 04). I adapted Bruce's recommended approach for use in SLX. In SLX, a user can request collection of so-called SOB statistics. (My choice of nomenclature is not a reflection on Bruce; it's shorthand for Schmeiser's Overlapping Batch Means method. Users remember catch phrases such as SOB.) SOB output for an estimated mean value is as follows:

- An estimate of the error is shown.
- Significant digits are underscored.
- The first questionable digit is not underscored.
- An "X" is shown to the right of the questionable digit.

The following is an example of SOB output:

```
SOB(Time in System)

     ~Value              ~Error        Significant Digits
   ~15.7295             ~0.0120              15.72X
```

Complexity Reduction 4 – Teach, teach, teach!

Those of us who develop models or modeling tools for others are often placed in the position of having to educate our users. Teaching is one of the most effective tools for helping others deal with complexity. To maximize the effectiveness of our teaching, we must optimize our teaching techniques around the ways in which people learn. In the paragraphs that follow, I will devote considerable attention to this topic, because of my passion for the subject.

Professorial egos notwithstanding, people are least likely to remember that which they only *hear*. The obvious corollary to this fact is if you're going to say things and expect your students to remember them, you must say them in memorable ways. Let me illustrate by example. Years ago, I was in charge of tech support at the University of Michigan Computing Center. In those days, all freshman engineering students were required to take an introductory computing course called Engineering 102. When the 900 or so 102 students were assigned a difficult problem, they descended on us in great numbers. The good students came well in advance of assignment due dates. Poor students waited until the last day, and many of them thought they could trick us into doing their work for them. One such fellow came to me after a 45-minute wait in line and said "I'm a 102 student. You may be familiar with our assignment." My eyes rolled over several times. "I just don't know where to begin," he continued. I thought for a few seconds and told him "When you do, come back." He was stunned. I told him that had he asked me to compare the merits of two or three possible solutions he had devised, I would have talked to him all day, but that I wasn't going to do his job for him. Finally he asked "When I come back, may I come to the head of the line?" I thought for a second and responded "No!" From time-to-time I reflect on this incident. I imagine that the student is now a grandfather. I envision his grandchild coming to him and saying "Grandpa, will you help me with my algebra homework? I have a really hard problem, and I don't know where to begin." "When you do, come back" he responds, and the torch is passed to a new generation. If you're going to say things and expect people to remember them, do so in memorable ways.

People are considerably likelier to remember that which they *read*, in part because they can reread it. In the simulation community, we are fortunate to have excellent textbooks, but in the wider world of computing, writing skills are remarkably lacking. The worst parts of my day are spent wading through documentation produced by marginally literate 22-year-olds. Reading poorly written descriptions is like swimming in a cesspool; it's difficult and most unpleasant. Here are some examples:

- "This function returns zero if there are no errors." (And if it fails, what happens?)

- "Invalid argument." (to an 8-argument function)

- "If the problem persists, contact the person who manages your network." (I manage my own network.)

- "You often will want to see what's happening inside your skin. You can do so through a text control or through a log file. You can create a TEXT element and place it on part of your skin temporarily."

To encourage good writing on the part of our students, we must set good examples in our own writing. A physics professor once posed the following problem on an exam: "Explain how you could determine the height of a tall building using a barometer." Anyone who's ever flown in an airplane knows that barometric pressure decreases with altitude. In theory, accurately measuring the decline in barometric pressure when moving from the bottom to the top of a tall building would allow us to determine the building's height. This is the answer the professor expected to get to his poorly stated question. One student answered as follows: “Go to the top of the building. Tie a long rope around the barometer and lower it to the ground. The length of rope required is the height of the building.” The student felt he deserved full credit for his answer, and the professor thought the student deserved no credit. Another professor was brought in to arbitrate. In the negotiations that ensued, the student presented *many* solutions to the problem, including the following:

- Go to the top of the building and drop the barometer. Measure the time it takes for the barometer to crash into the sidewalk. Assuming constant acceleration, this time can be used to determine the building's height.
- Tie a long rope around the barometer and use it as a pendulum. The period of a pendulum is proportional to its length.
- Go to the basement of the building and find the office of the building superintendent. “Do you see this wonderful barometer? If you'll tell me how tall the building is, I'll give it to you.”

We have no right to expect good writing from our students if our own writing is poor.

We *must* insist on good writing from our students. A number of years ago, I taught at the Northern Virginia campus of Virginia Tech in the Washington, DC area for five years. At first, I was fond of using essay questions on my exams. I quickly discovered that if you ask a question requiring a black or white answer, poor students will give you a gray answer, often taking many pages to do so, in the faint hope of receiving partial credit. Ultimately, I devised the 50-word response essay question. Answers of more than fifty words received no credit. Consider the following question: “What is a marriage?” Answering that question in fifty words or fewer requires getting immediately to the point. I always told my students “When I ask you to explain X, the first two words of the A answer are 'X *is*.' If you begin by saying 'X *involves*, you're working on a B+.” When I used such questions in a class with 25 students, I could count on getting 23 “X is” answers. Of the remaining two students, one would think I was kidding, and the other just didn't get it.

As one of my mentors, William D. Revelli used to say, “The teacher’s responsibility ends not with the telling, but the doing.” Students are a captive audience. Use every opportunity to insist on good writing. Set the bar high. Years later, they'll thank you.

Finally, people remember best that which they *discover*. The teacher's challenge is to pose problems that are neither too hard nor too easy. If a problem is too hard, the teacher ends up having to tell the student the solution, and if a problem is too easy, the student already knows the answer (Polya 45a). In both cases, no discovery takes place. Plato said “Ideas must be born in the minds of students. We teachers are merely the midwives.”

Providing tools that aid discovery is a great way to cut through complexity. Allow me to illustrate. A number of years ago, I helped a small company that was developing software for inventory management. When this company dealt with prospective customers, they used the following approach:

- First, they'd ask the customer to give them an estimate of monthly inventory-related costs.
- Next, they'd informally introduce the notion of a probability density function by drawing a bell-shaped curve and explaining that monthly costs were subject to random variation, with certain ranges of values likelier than others. A customer didn't have to understand integral calculus to get the gist of the explanation.
- Next, they'd draw another bell-shaped curve that was taller, narrower, and to the left of the original curve. They would explain that using their software would lower average costs and by imposing better controls, decrease the variation in monthly costs.
- Finally, to underscore the notion of randomness, they would point out that the right tail of the improved distribution overlapped the left tail of the original distribution, explaining that there would occasionally be months in which new strategies turned out to be inferior to old strategies, but in most months the new strategies would be superior.

This was a very nice way of explaining the complexities of randomness; however, the tools used to display model results provided an even better way. Outputs were presented to the customer as follows:

- Many replications were run, and a distribution was fitted to monthly costs.
- The distribution was displayed and could be traversed with a “hot” mouse. The hot mouse displayed the cost and probability split, e.g., “60/40: $12,000,” at any point along the fitted distribution. In many ways, this was a cheap trick, but it very quickly communicated the concept of a PDF. Customers would start by wiggling the mouse around the mean of the distribution. (“I see that the 50/50 point is around $10,000.”) Invariably, their next action was to explore the right tail of the distribution. (“Whoa! There's a 10% chance my costs could exceed $15,000. How could that happen?”)
- The answer to this question could be obtained by right clicking, which would display a column vector of the components of cost taken from a representative replication. Customers could examine cost components and decide which seemed out-of-line. (These were not necessarily the largest components, which may have been fixed costs.)
- Right clicking on an individual cost component brought up a row vector describing the history that lead to the cost in the selected replication. (“I see that we used a large number of X parts one month, subsequently ran short of them, and had to air freight X parts from another supplier. We always knew this was a problem, but I guess we underestimated its importance”)

Learning by discovery is a great way of cutting through complexity.

Complexity Reduction 5 – Provide easy access to true primitives.

SLX's “wait until” was described above. it is a good example of a true primitive. It cannot be further decomposed into lower-level primitives. Furthermore, there's no need for a software developer to “hard wire” higher-level variations of wait until, because wait until is exposed and easily exploited by end users.

Consider the one-line single-server queuing model (the grandfather of all queuing models). How do we specify the conditions for model shutdown? In a naïve model, the system shuts down at a specified time, ignoring whether the server is busy or anyone is waiting. In a more sophisticated model, a hybrid condition is used, e.g., “wait until shutdown time is reached, *and* the server is idle, *and* the queue is empty. Consider how we could model a customer going to his bank, getting in line, but reneging if it takes more than five minutes to be served. We could save the current time plus five minutes in a variable and then have the customer “wait until time attains the value of this variable *or* he/she is served.”

If true primitives are hidden inside impenetrable black boxes, users are constrained to follow the paradigms implicit in the black boxes. Freedom of expression is inhibited, and unless a universal collection of black boxes is provided, some requirements may go unfulfilled. Primitives should be made as small as possible, and as general as possible.

Complexity Reduction 6 – Build orthogonal features that are easily combined.

The GPSS language has servers, and it has arrays, but it has no arrays of servers. The concept of a server is familiar to all simulation practitioners, and the concept of an array is familiar to anyone who knows how to write a program. Combining these two orthogonal concepts ought to be easy. The worst possible cure to GPSS's deficiency would be to provide “server arrays” in which the two concepts are inseparably bound together.

In our previous discussions of SLX's wait until feature, we have ignored how one identifies state variables in a model. In SLX, state variables are called control variables, and they are defined by prefixing their declarations with the word “control.” This prefix tells SLX's compiler to insert tests for non-empty state change queues at each point a control variable is modified. A control prefix can be applied to any SLX base type. Control integers and control Boolean variables are commonplace, but one can also use control floating point variables or even control character strings. The key points are that “control” is orthogonal to the other aspects of variable declaration, and “control” is easily added to any declaration.

In operating systems and in some modeling tools, state variables take the form of stylized, built-in data types. For example, GPSS includes so-called “logic switches,” and operating systems include semaphores. Both are used to achieve required synchronizations. Black boxes of this sort unnecessarily limit a programmer's freedom of expression.

Complexity Reduction 7 – Use formal methodologies, but only where appropriate.

Many older simulation languages are truly ugly. A major contributing factor to their ugliness is the failure of their developers to use formal grammars to define language syntax. I have been in simulation for roughly 39 years. Prior to giving my Titan talk, over that 39-year period, I had talked with exactly one simulation tool developer who was conversant with formal grammars. Grammars have been used by knowledgeable language designers and compiler writers since the 1950s. The failure to use a grammar in a modern product is inexcusable. After I gave the Titan talk, in which I talked about the use of grammars, a member of the audience came up to me and told me that my discussion took him back to his student days, when he had done research on the use of grammars. He became the second simulation professional with whom I had a meaningful conversation about the use of grammars.

Formal methodologies exist for other aspects of developing simulations, e.g., Nance's Conical (not *comical*) Methodology. However, some methodologies impose limits that are unacceptable for general-purpose modeling. For example, Petri Nets can be used as the basis for building models. One advantage of using Petri Net methodology is that it makes it possible to formally analyze models and prove certain properties of them. The major disadvantage is that the technology scales poorly. Petri Nets are too cumbersome for modeling very large simulation applications, e.g., simulation of air traffic control or telecommunications network protocols.

Complexity Reduction 8 – Provide open architectures.

You might be surprised to learn that the SLX kernel does not include built-in queues or servers. These common simulation entities are constructed at higher levels of SLX, by using macros, user-defined statements (a sort of “super macro”), and run-time libraries. Wolverine Software provides a subset of GPSS implemented in SLX. One of the major differences between GPSS and the SLX-hosted implementation of GPSS is that the SLX user can get inside the implementation of GPSS statements, which are black boxes in GPSS. For example, a user can actually step through the logic by which a server is acquired. While most users never experience the need to explore implementation at this level of detail, there are two circumstances in which it is useful to do so.

First, there may be occasions in which the operation of a complex statement doesn't match a user's expectations. By carefully examining the inner workings of a statement, any questions can be answered. Second, there may be occasions when no building blocks in the Wolverine-provided collection exactly meet a particular modeling requirement. If one building block comes close, examining how it works is a good way of starting the process of building a customized feature that exactly meets the modeling requirement.

I'm a great believer in open source code, but only up to a point. Most aspects of compiler implementation and many aspects of advanced run-time support fall in the province of the expert. I believe that these areas can be confined in a small, “untouchable” technology kernel, and that all other technology should be as exposed as possible.

Complexity Reduction 9 – What's useful to the developer may be useful to the user.

Simulation packages are characterized by heavily used, sizeable run-time libraries. The presence of large numbers of library functions leads to a larger number of function usage errors than might be found in other types of software. A user who is confused about the arguments used to sample from a given statistical distribution might inadvertently specify a negative shape parameter. When such errors are diagnosed during model execution, it is extremely helpful to issue diagnostic messages at the point of function invocation, e.g.,

```
y = -1.0;
x = rv_gamma(Service, 10.0, 1.0, y);       (“y” is highlighted in red.)
```

```
•• Execution error at time 0: gamma distribution: shape must be greater than zero
```

SLX includes a "diagnose" statement for issuing such messages. It is heavily used within SLX itself. It became apparent early in the development of SLX that there was no good reason to restrict use of the diagnose statement to "system" use. This is an example of a feature developed for the developer that turned out to be useful to everyone.

Another example is SLX's "describe" statement:

```
int         PeakSize;
describe  PeakSize "Maximum number of requests in the server queue";
```

The information provided above is displayed when a user right clicks on PeakSize and then selects "About PeakSize" in the menu that pops up. This little jewel took about an hour to develop and has proved to be extremely useful to both developers and users.

Complexity Reduction 10 – Provide *extensibility* mechanisms.

Software kernel functionality should be as small as possible, but as general as possible. Trying to provide all things for all people at the kernel level is an extremely bad approach. Notwithstanding this obvious principle, "feature wars" rage on between competing software vendors. (As an aside, this is one way in which software developers differ from academics. An academic will lean over backwards trying to distinguish his work from that of others. If a software developer sees a great feature in a competing product, "me, too" prevails.)

If kernel logic is kept small, the goal of high applicability can be attained only by including tools that make it easy to extend kernel functionality to widely varying applications. Extensibility mechanisms are the appropriate vehicle. Let me illustrate.

One of my SLX users once contacted me when he faced the following problem. He had data that was most naturally described using a 3-dimensional array. Unfortunately, he needed a 1000 x 1000 x 1000 array, and this was too large for available memory. Had he been able to store his data in an array, the array would have been very sparsely populated. My enthusiasm for adding sparse array support was low, and his enthusiasm for doing so himself was even lower.

I had previously done some work with associative mappings. The classic example of an associative mapping is a telephone book. Given a first name and a last name, a telephone book provides the telephone number associated with a person. I was immediately drawn to the possibility of using an associative mapping to implement sparse array support. In general terms, my customer wanted to find the floating point value associated with an (i, j, k) 3-tuple of integers.

The great mathematician and teacher George Polya (Polya 45b) described what he called "the inventor's paradox:

> When trying to solve a difficult problem, sometimes the more ambitious plan is likelier to succeed than the less ambitious.

As an aside, allow me to observe that we computer scientists tend to be overly fond of devising ambitious plans. As the saying goes, "Ask a computer scientist to build you a toaster, and he'll come back the next day and tell you 'You're thinking too small. A toaster is really an instance of a breakfast food preparation device.'"

I began to consider my customer's problem in very broad terms. What would it take to associate an output value of any given type with a collection of N input variables of any given types? I refined my requirements to the following list:

- I needed to be able to define associations.
- I needed to be able to create and destroy individual instances of associations.
- I needed to store collections of association instances.
- Most importantly, I needed to be able to rapidly retrieve the output value of an association instance.

My next step was to divide and conquer. I focused on the first two requirements and ignored the last two. Many of us are first exposed to this technique when we learn geometry (Polya 45c). When asked to find the point that satisfies two conditions, we concentrate on the first, and ignore the second. For example, all the points that satisfy the first condition may lie along a line that we can construct. Ignoring the first condition and concentrating on the second, we may find that all the points satisfying the second condition lie along an arc we can construct. Presto! The point lies at the intersection of our constructed line and arc.

The natural way to define an association is as an object class containing N input values and their associated output value. Clearly, I could create and destroy instances of such a class. Returning to the remaining requirements, I considered how I could store a collection of instances of an association. SLX provides *sets* for exactly this purpose. I could place association instance objects into a set, and if I needed to destroy an association instance object, I could simply remove it from the association's set. This left the final and most problematical requirement. How could I rapidly retrieve association instances? SLX includes a *retrieve* verb that is used to retrieve members of a *ranked* set. I could define the set as being ranked on the input variables. Retrieval of objects from ranked sets is done in SLX by using a variant of Henriksen's Algorithm (Fishman 00) for the event set. It is very fast.

At this point, I had satisfied all four requirements. My solution used features of SLX that were accessible to any user; however, it required very careful programming, so I was unhappy with its ease-of-use. I could have quit at this point and dumped everything back in my user's lap. (After all, having bad breath is preferable to having none at all.) I began to contemplate ways of relieving my customer of as much of the programming burden as possible. SLX includes a very powerful macro capability, and it includes the ability to define entire statements. (User-defined statements can be regarded as a sort of super-macro.) Without going into detail, let me provide an example of the end result:

```
typedef enum { RED, GREEN, BLUE }  color;

association(int, double, color : string(20)) idc_to_string;

procedure main()
    {
    string(20)          s, joh;

    associate "Jim Henriksen" with 1000, 2.5, BLUE in idc_to_string;

    joh = map_idc_to_string(1000, 2.5, BLUE);

    print (joh)         "String \"_\" is associated with 1000, 2.5, BLUE\n";

    remove_association(1000, 2.5, BLUE) from idc_to_string;
    }
```

The example above shows an association that maps an (integer, double, color) 3-tuple into a character string. "association" is a user-defined statement. (In this case I happened to be the user who defined the statement, but anyone else could have done the same.) "associate" is a defined statement that creates an instance of the defined association. "map_idc_to_string" is a macro that is defined in the expansion of the "association" statement. One might describe "association" as a meta-macro, since it in turn generates other macros. Finally, "remove_association" is a user-defined statement that does the obvious.

My example illustrates just how far you can go when you have the right collection of primitives and extensibility mechanisms to combine these primitives in creative ways. The example transforms a very hard problem into a simple problem. This is reduction of complexity.

Complexity Reduction 11 – Build interfaces at the proper level.

When interfaces are built at too low a level, users are forced to deal with more details than they would prefer. When interfaces are built at too high a level, users are constrained to follow any paradigms the developer has built into the interfaces. Examples of both extrema are commonplace.

A second way of saying the same thing is “find the right division of labor between the developer and the user.” Let me explain what I mean in memorable fashion. One of the recently discovered side effects of Viagra® is that it causes disturbances of vision in a small percentage of men. Increased libido and decreased acuity of vision are a colossally bad combination. (Think about that for a moment.) In the realm of software, we face a similar problem: who provides the power, and who provides the vision? When a developer tries to do both, the user is constrained to follow the developer’s vision of how a particular problem should be solved. A friend of mine is fond of saying “We’re in the business of inventing new paradigms. We can’t do that if we have to use someone else’s.” I believe that developers should provide the power (the tools that would be difficult, if not impossible, for users to build on their own), and users should provide the creativity. I call this the *Viagra Principle*.

Yet a third way of saying the same thing is “Be sure that transferal of anxiety flows in the right direction.” Transferal of anxiety is an interesting concept. Let me illustrate. From time-to-time, I’m contacted by a customer who has bought a new computer. He/she says “I replaced my old 1 GHz laptop with a new 3 GHz laptop, but your software doesn’t run three times as fast. What’s wrong?” It’s not my problem that he/she bought a lousy laptop; it’s his/her problem. However, he’s/she’s trying to transfer his/her anxiety to me. To deal with such situations, I have a program that thoroughly tests a CPU’s computational power. Running this program on two different machines quickly reveals that in some areas one machine outperforms the other, while in other areas, the converse is true. I send a copy of this program to the user and tell him/her to run it on both computers and compare the answers. He/she goes off and spends three hours doing so. I have reflected anxiety back in his/her direction. After three hours, he/she concludes that the only way to compare the speed of two machines is to benchmark important applications, which is exactly what he/she should have done in the first place. This is transferal of anxiety in action.

People are least likely to remember that which they are only *told*. They rarely forget that which they *discover*. This example illustrates that principle.

Over the years, I have observed that most users accept more anxiety than they should. Many, many times, I’ve been contacted by users who have suffered with a software glitch for a week that I’m able to correct in an hour.

Complexity Reduction 12 – Build swimming pools, not wells, vast ponds, oceans or tarpits.

I once gave a talk to a group of GPSS users. I explained that implementing a language as large as GPSS is like constructing a vast pond. One user replied “I like that pond. I can get around in it without fear of drowning.” Let’s expand on this analogy.

Special-purpose (vertical market?) applications take the form of wells. The advantage of a well is its depth. All pertinent issues are dealt with within the confines of the well. The disadvantage of a well is that different problems require different wells.

I’ve already stated the advantage of vast pond architecture. The disadvantage of this architecture is that one must visit many regions of the pond to tackle complex tasks.

The best compromise is to build swimming pools. The only disadvantage of swimming pool architecture is that users must know how to swim. However, ultimately, swimmers are survivors. It's impossible to swim in wells and shallow ponds.

Large organizations have the resources to attempt to build oceans, in which entire *ships* can sink. Small organizations do not. Small organizations are limited by the amount of complexity they can handle in their development efforts. The products they develop are inevitably of higher quality than those of very large organizations, because their developers have great respect for the complexity dragon. They go to great pains to avoid foisting complexity on their users. Relatively unfettered large organizations often end up producing tarpits rather than oceans. Does this strike a chord?

Conclusions

- Complexity is public enemy #1.
- All simulation professionals are in the business of reducing complexity.
- I've presented twelve approaches to reducing complexity.
- I hope that you can make use of these approaches.
- Unlike Saint George, we may be unable to slay the complexity dragon, but we may be able to tame him.

Acknowledgements

I'd like to thank the WSC 2006 committee for honoring me as the 2006 "Titan of the Industry." I'd also like to thank Tom Schriber and Deb Medeiros for proofreading this paper.

References

Fishman, G.S. 2000. *Discrete-Event Simulation,* 197-211. New York, NY: Springer

Henriksen, J.O. 1975. Building a Better GPSS. In *Proceedings of the 1975 Winter Simulation Conference* (Out-of-print)

Polya, G. 1945a. *How to Solve It*, 1. Princeton, NJ: Princeton University Press.

Polya, G. 1945b. *How to Solve It*, 121-122. Princeton, NJ: Princeton University Press.

Polya, G. 1945c. *How to Solve It*, 81-84. Princeton, NJ: Princeton University Press.

Schmeiser, B. 2004 Simulation Output Analysis: A Tutorial Based on One Research Thread. In *Proceedings of the 2004 Winter Simulation Conference*, Eds. R.G. Ingalls, M.D. Rossetti, J.S. Smith, and B.A. Peters, 162-170. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Author Biography

Jim Henriksen is the president of Wolverine Software, which he founded in January, 1976. Jim has presented many Winter Simulation Conference papers. He was one of four joint keynote speakers at the 25th anniversary conference. He has served as business chair and general chair. For a period of five years in the early 80s, Jim taught at the Northern Virginia extension campus of Virginia Tech.

Jim's major interests are the construction of compilers and run-time support for discrete-event simulation and animation. He has had partial or total responsibility for developing eight compilers and five generations of animation software.