Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

SOFTWARE ENGINEERING APPLIED TO DISCRETE EVENT SIMULATIONS

Kenneth N. McKay
John A. Buzacott
John B. Moore
Christopher J. Strang
WATMIMS Research Group
Department of Management Sciences
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1

ABSTRACT

Developing simulation programs shows many similarities with classical system software development tasks. In simulation one is often concerned with allocating and deallocating resources. Two forms of deadlock -- the 'deadly embrace' and 'aprés-vous' -- can be troublesome to simulators unless they know how to avoid them in the first place. Critical races and time dependent functions are other characteristics shared between simulation and systems programming. If simulation is viewed and taught as data processing, the simulator will be ill-prepared for writing simulation code.

Most simulations in industry are not written by skilled software developers. There is good reason for this: the simulation writer must ultimately understand the problem, its features and the managerial concerns that lead to the requirement for the simulation. However, simulation writers with no software training are poorly equipped for developing simulation code that is easy to design, debug, verify, maintain and explain. A number of simple modern software engineering techniques which are described in this paper can be applied to simulation programs in a practical way to improve both the quality of the simulation and the productivity of the simulator.

1. INTRODUCTION

There exist four major concerns in simulation. These are: input and output data analysis, model verification and validation, communication, and model construction. Software engineering [ZELK79] concepts can simplify verification, lessen the impact of communication difficulties, and ease the burden of model construction. Specifically, the aspects of software engineering used in systems programming are directly applicable to simulation development [RYAN79].

The analogy between simulation and systems programming is discussed in section 2. This is followed by a description of software engineering techniques as applied to simulation development. Concepts are illustrated using a model of a surface mounted technology (SMT) printed circuit board assembly line that was developed in SIMAN [PEGD85]. This assembly line model is a discrete-event simulation using SIMAN in its block form with no special FORTRAN code.

2. SOFTWARE ENGINEERING AND SIMULATION

Software engineering encompasses many aspects of software development: requirements, specification, design, coding and debugging, performance evaluation, verification, validation, project management, documentation, communications, standards, migration, portability, security, etc. In the late sixties and early seventies, landmark work such as Dahl/Dijkstra/Hoare [DAHL72], Brooks [BROO75], and Knuth [KNUT68] created a significant interest in academic and industry circles concerning the method and madness used to develop software. Research concentrated on operating systems, languages, design methods, databases, verification techniques, data processing applications, etc., with the majority of work being performed in computer science groups and defense agencies.

Although surrounded by clouds of mystique and magic, simulation development is essentially a software development project. An additional difficulty not mentioned in the introduction is that many programmers developing simulations are not trained in software development practices which span requirement analysis through verification. Coming from engineering or the sciences, the developers may understand what needs to be simulated but not how to go about doing it. They might have had one or two courses in FORTRAN, COBOL, PASCAL or ASSEMBLER.

Assuming that a simulation writer wants to have an easier time of it, where does he/she start? As with any set of concepts and tools, there are some tools more appropriate for certain jobs than others. The size and category of job may dictate the tool selection. The reader of software engineering literature should always keep the problem and solutions in perspective; the solution is the tool, not the end objective.

For our purposes, we rule out the large-scale simulations of the defense and aerospace industries. Sheppard [SHEP83] provides a good overview of the software engineering aspects that can be applied to large simulations. Of the remaining simulation projects almost all appear to be one or two people projects and contain less than 5,000 lines of code. The simulation projects entail resource control, resource allocation, process flow, and timing considerations. With few exceptions, the common simulation languages are low level and the programmer is required to

build up primitive functions that relate to the concept to be modelled. These problem characteristics, potential solutions, and development tools are identical to small system programming tasks. In a high-level tutorial survey of software engineering and simulation [RYAN79], the usefulness of operating system research to simulations is noted. We stress the similarity in the software and delve deeper into the areas of software engineering that can benefit simulation projects.

A software project consists of several phases. The first is the requirements analysis and problem definition phase. The design phase is next which is followed by the detailed design and implementation (programming). Software is then exposed to a series of unit, module, and systems tests to verify that the requirements are satisfied. Finally, the customer obtains the software to see if it really does the job. Early customer involvement is crucial with software projects. A prototyping approach can facilitate this interaction. A paper by Wolverton [WOLV74] mentions a common rule of thumb that indicates that 40% of development cost will be in analysis and design, 20% in coding and debugging, and 40% in the checkout and test phases. While each project will vary somewhat, sufficient time must be given to the front-end work. Details of each phase are discussed in the following sections.

3. REQUIREMENTS ANALYSIS

The requirements phase is one of the most important and difficult phases of a simulation project. The issues include: what is to be modelled; what are the main components in the model and their characteristics; what questions are to be answered; what are the assumptions; what are the limitations. Duket [DUKE82] emphasizes that the purpose of a simulation is to answer questions and influence decisions. Note also that, what will not be in the simulation is often as important as what is!

A major task is to determine how to communicate the information among the people involved in the project. There has been significant research into formal specification languages and how to minimize the errors introduced at this point of the project [BALZ85]. Outside of the defense and related fields, there have been few reported efforts of using formal specification languages in industry. There are some problems with the formal specification language approach. First, a number of the languages have been algebraic in nature for conciseness and provability. This makes expressing meaning and intent difficult. Second, if the language is not algebraic, it still has a syntax and vocabulary which does not solve the communication problem between the analyst and 'end user' who needs to understand what the analyst writes in the functional specification. An ideal specification language would be one in which the 'end user' can describe their desires with ambiguity, missing data, conflicting data, and erroneous data. That is one in which the language system knows what the person really meant. We are not aware of any such language or system. Even with rigorous mathematical notation, such Artificial Intelligence systems are research concepts and experimental in nature [GOLD86].

Assuming that a suitable specification language does not exist, the simulation developer must rely on the traditional methods of requirements analysis. Unfortunately as one performs a requirements study, one learns that the popular saying: "constants aren't and variables won't" is true. The state requirements will be ambiguous, incomplete, conflicting, wrong and unstable. To address these issues, the analyst is required to know about simulation development as well as the real world being modelled. Interpretations, translations, and definitions are important parts of this "black art" of determining the requirements.

In many ways, this phase is similar to the task faced by a systems programmer when designing a support tool such as an operating system. The skilled system programmer will listen to the 'end user' and mentally build a 'model' consisting of application specific information and support functions. For example, a particular machine to place paste on a board has application specific information: it processes computer cards, puts paste on specific spots, etc. It also has support or operational information: it processes one thing at a time, takes a certain amount of time for this action, cannot move its object until the next process downstream is free, etc. Clearly, there is application information and what can be called generic information.

To illustrate the concepts of abstraction, Figure 1a lists the function of each segment of the SMT line. In the seven segments, there are sixteen unique processes that are modelled with six abstract structures (Figure 1b). Figure 1c shows the first two segments and how the abstraction matches the initial description.

The separation of the problem into application specific and generic characteristics can be accomplished in simulations and should be consciously performed. Documenting this separation in the requirements document is one way to minimize the impact of poor communications. The underlying structure will be clear as will the areas of application (which change with time).

1	prepping
2	screening and paste application
3	top SMT device population
4	vapor-soldering
5	pin-through-hole device population
6	bottom SMT device population
7	wave-soldering

Figure 1a: Seven Segments of the SMT Line

Processes/Machines	Abstractions
1 Magazine Type A/B Load	1 Magazine Load
2 Magazine Type A/B Unload	2 Magazine Unload
3 Wave Solder	3 Single Panel Machine
4 Cure Oven	4 Multiple Panel Machine
5 Panel Inversion	5 Inspect - Inline Rework
6 Top SMT/PTH Placement	6 Inspect - Parallel Rework
7 Top Nonstandard SMT/PTH Placement	
8 Bottom Placement	
9 Aqueous Cleaner	
10 Paste Dry	
11 Vapor Phase Reflow	
12 Screen and Paste	
13 Inspect - Inline Rework	
14 Inspect - Parallel Rework	
15 Pre-Clean	
16 Solvent Wash	

Figure 1b: Processes and Abstractions

Processes/Machines	Abstractions
1 Unpack	1 Single Panel Machine
2 Pre-Clean	2 Single Panel Machine
3 Magazine Unload	3 Magazine Unload
1 Magazine Load	1 Magazine Unload
2 Screen & Paste	2 Single Panel Machine
3 Inspect - Inline Rework	3 Inspect - Inline Rework
4 SOIC Placement	4 Single Panel Machine
5 PLCC Placement	5 Single Panel Machine
6 SMT Non-standard Placement	6 Single Panel Machine
7 Inspect - Parallel Rework	7 Inspect - Parallel Rework
8 Paste Dry	8 Multiple Panel Machine
9 Magazine Unload	9 Magazine Unload

Figure 1c: First Two Segment Analysis

4. DESIGN

Once the requirements of the simulation are known and agreed to, the design problem can be tackled. The issues in a simulation design are: what philosophy or approach should be used in the design; what is a suitable architecture for the model; what is the support structure; and how should the design be documented. The design of the model should not be confused with the code design and what is internal within the software. The architecture is concerned with the interfaces -- shape, form, and other characteristics. The design should proceed relatively independent of any simulation language you choose. Most provide the necessary features in one form or another.

During the past decade, there have been a number of software design methodologies (or mythologies if you wish) developed and widely used by software developers. A recent survey by Yau [YAU86] provides a good overview of the most popular: HIPO, SADT (Ross), Structured Design (Yourdon and Constantine), Composite/Structured Design (Myers), Module Interconnection Languages (DeRemer and Kron), Jackson's Design Methodology, Warnier's Design Methodology, Stepwise Refinement (Parnas), Data-Oriented Design Techniques, and Object-Oriented Design Methodology. All of the methods provide frameworks within which the problem is to be analyzed and the design annotated. Each results in a slightly different design and orientation which might not be obvious upon first glance but which appears later when functionality is changed or process characteristics altered. It is quite possible that the largest benefit of design methods lies in forcing developers to approach the task systematically. A constant danger is the assumption that one design technique will work for all problems and further that a finished documented design is a good design. The fallacy of assuming that the design process can be totally mechanized and systemized is pointed out in [PARN86]. The data structure (Jackson, Warnier), data-oriented, and object-oriented methods appear to us as the most relevant to simulation development. Several excellent texts on design for reference are [JACK75], [MYER79], and [YOUR79].

While it may be desirable to understand the above design methods and intuitively grasp their subtleties, the finer points of good design come only through practice. Based on our experience, the heuristics listed in Appendix A are usually sufficient to give a developer both the structure and orientation needed to design either systems programs or simulation software. Another approach to problem decomposition is [COHE82]

which involves decomposing the model into a series of queues and their connections.

The structure, orientation, and design methods do not delete the requirement for the designer to understand what is to be modelled, what forms of architecture are possible, and what support techniques can be used. For instance, there are many design issues independent of the notation technique. Should the flow be interrupt driven or polled? Should the design protect against deadlock situations during resource allocation, and if so, are semaphores appropriate? How is error recovery to be performed without contamination of the environment? How can critical races be prevented? These problems and corresponding solutions have been studied by computer scientists for many years. If simulation developers are not aware of issues such as the deadly-embrace and after-you deadlock syndromes, then the wheel will be re-invented time and time again through painful and frustrating lessons. Textbooks such as Shaw [SHAW74], Per Brinch Hansen [HANS73], and Freeman [FREE75] provide excellent information on these and related issues. A short course on operating system internals would benefit almost every simulation developer and enable them to identify situations and understand what can be done.

5. PROTOTYPING

Prototyping is the development of a small experimental version of the final software and is often considered too expensive. However, correcting ambiguities and misunderstandings at the specification stage is significantly cheaper than correcting a system after it has gone into production as pointed out by Gomaa and Scott [GOMA81].

Prototyping provides many benefits to the 'end user' and modeller. First, a prototype provides feedback to the modeller and 'end user' that the system is understood and can indeed be modelled. Second, it shows the 'end user' what information is required to feed the model and what the output of the model will be. Third, the prototype provides early visibility as to development problems and serves as an excellent base for estimating the resources required to complete the model.

A simulation prototype should contain at least one instance of all unique logic that will be found in the final model. This is determined by studying the significant components and their interfaces which were documented during the design cycle. A final model will often have many similar elements, which for the purpose of illustrating a concept, do not have to be included in the prototype. A prototype should be significantly smaller than the final model and as a result is faster and easier to code, debug, and verify.

In the seven segment SMT line, there are twenty-one discrete processes and thirteen magazine load and unload points. Figure 2 shows the prototype model structure that was used to show that the six abstract building blocks were sufficient to perform the modelling and that SIMAN was a reasonable choice for executing the model. The inline inspection was trivial and not included in the prototype. The prototype had the necessary logic for multiple panel types, segment setup and flushing between panel types, operator allocation, shifts, machine breakdowns, batch splitting in the event of segment failure, and magazine allocation. The prototype included the statistics for downtime, blocked operation, setup, and processing analysis. The initial prototype was approximately four hundred SIMAN statements.

As time progressed, the prototype was expanded to include different styles of multiple panel machines, single machine segments, and transportation requirements between segments. The prototype was used to train and instruct students and industry personnel on the model requirements and SIMAN.

Segment 1

1 Single Panel Machine
2 Inspect - Parallel Rework
3 Magazine Unload

Segment 2

1 Magazine Load
2 Single Panel Machine
3 Multiple Panel Machine
4 Magazine Unload

Figure 2: Prototype Structure

A developer should not view a prototype as a one-shot throwaway. If the modelled system does not exist, there will be many variations and design changes as time progresses. A prototype can be used to design and test the change without involving the complexities of the complete model. Further uses of a prototype are in training new users of the model; training new programmers in the language; and as a demonstration tool. It is easier to understand and experiment with a model of four to five hundred lines of code compared to one of five thousand lines.

6. DETAILED DESIGN

In Yau's review [YAU86], several detailed design techniques and representation methods are described including structured programming, flowcharts, Nassi-Shneiderman diagrams, heirarchical graphs, and program design language. Any of these methods and some others such as data flow diagrams or finite state automata [ZELK79, HOPC79] can be used to document the detailed design. Not all of the methods provide the same levels of comprehension to a reviewer or maintenance person [SHEP81] and care should be taken in documenting the concepts, exceptions, and critical assumptions.

Our experience with systems programming and simulations has led us to favor the data

flow, finite state, and program design language approaches. Often all three methods are used on the same project. We have chosen the three methods partially because of the similarity of simulations to operating systems. A major similarity is that systems are control block driven with the blocks being totally self-explanatory as they flow through the system. Model entities can be viewed in the same light. The data flow and finite state diagrams can be useful for identifying the flow and necessary control information, while the program design language is excellent for describing the semantics of what is happening. A possible danger of the graphical approach to documenting a design is the tendency to draw a pretty picture -- changing the design to be visually pleasing. This does not always guarantee a good design.

An important point about detailed design is the need for an intermediate layer between the actual programming language and the high level architecture. The layer provides the opportunity to have internal algorithms designed and written without the encumbrance of language syntax. The program design language and information on the diagrams should not be at the statement level of detail. The intent and meaning of the logic should be conveyed, not the syntax. The designer should be careful to highlight the exceptions and the key steps and not overwhelm the design description with trivia.

Figures 3a and 3b illustrate how an English description can be combined with pseudo-code to document the detailed design of a Single Panel Machine.

```
There are two sets of logic required for single panel machines.  The first is when
another machine follows and the second for when the next structure is a magazine
unload point.

Both forms are identical in how operators, processing, breakdowns, line flushing,
and blockages are handled.

    Before the machine can be considered 'obtained', the operator (if needed)
    must be available and nothing currently in the machine.

    Concurrent logic (see breakdown logic) is used to time machine breakdowns
    and claim the machine with priority.  This eliminates the need for each
    machine to have breakdown logic inline.  The concurrent logic inserts a
    fake breaker part into the line to indicate that batch splitting should
    occur.

    The machine is considered blocked if it cannot seize the next machine/
    operator resource.

    Only after it successfully claims the next machine, will it free up the
    current one.

    When flush or breaker parts come down the line, no processing will occur.

If the machine is the last one in the line ....
```

Figure 3a: English Description - Single Panel Machine

```
At machine J-1
Seize machine J
    If unsuccessful, wait until machine J is free
If part is a fake one for breakdowns or line is being flushed
    Free machine J-1
Else
    If operator is needed for machine J
        Seize operator for machine J
            If unsuccessful, wait until operator is free
    Free machine J-1
    Delay for processing time
    If operator was obtained
        Free operator for machine J
Seize machine J+1
etc.
```

Figure 3b: Pseudo-Code - Single Panel Machine

7. CODING

Once the detailed design is done, the coding of the prototype can begin. There has been much written about GOTOless programming, structured programming, etc. [DAHL72]. Most of the literature makes sense when using a general purpose programming language. However, simulation languages are not general purpose programming languages and do not support all of the constructs and concepts found in languages such as PASCAL or MODULA-2. There has been some work [GOLD85] in identifying the software engineering requirements of simulation languages, but the research results have not migrated to the current languages. Writing simulation code in languages such as SLAM II [PRIT84], GPSS [SCHR74], and SIMSCRIPT [CACI83] is closer to writing in assembler than in Pascal. Some of the languages do exhibit a few features of FORTRAN. However, as a group the general purpose simulation languages are low level. There have been a number of special purpose languages such as AutoMod [AUTO86], MAP/1 [MINE86], and MAST [CMSR85] that are designed for a specific application area such as manufacturing and can be considered fourth generation languages.

Regardless of the language chosen, there should be standards imposed that will make the code readable, maintainable, and consistent. Kernighan and Plauger's book on programming style [KERN74] is a good starting point for individuals looking for guidelines. Structured walkthroughs are one technique for seeking defects and is recommended for all but the most trivial of simulations. Appendix B contains a list of our coding standards.

Simulation code can have a long life. If the code is consistent and follows a set of standards, it can be re-used in future simulations. This is especially true if the code is developed using 'black-box' concepts and clean interfaces. To gain this benefit requires discipline and the foresight to recognize that more than one simulation will be written. Building prototypes will help developers to design for reusable code.

8. VERIFICATION

After coding and initial debugging, the simulation code must be verified for accuracy. The question to be asked is whether the code reflects the descriptions found in the requirements document. This phase is the precursor to model validation. The validation phase determines if the behavior of the model adequately corresponds to that of the system being modelled. The verification phase on the other hand ensures that the model does what the user specified.

Verification can be viewed as rigorous debugging with one eye on the code and one eye on the model requirements. Chattergy and Pooch [CHAT77] recommend that simulation verification be performed in concert with the design effort using a top-down modular approach. Their approach tends to delay early visibility of a working model and does not incorporate a prototype model.

There are some cases of software development where formal specification languages can be used and rigorous verification can be accomplished [HAYE86]. Other automated tools have been developed to analyze software programs, identify areas of complexity, instrument the code to determine test effectiveness, etc. [CLAR76]. The automated tools have been oriented to the general programming languages and large-scale developments. Since simulation development does not fit into these categories, the developer must perform the verification manually. Unfortunately, programmers make poor testers of their own code [WEIN71] and must make conscious efforts to circumvent this problem.

One of the goals of verification is to show that all parts of the model work independently and together using the right data at the right time. A study of errors in system programs by Endres [ENDR75] classified problems found in a large system programming project and noted that approximately half the problems arose from requirements and the other half from the actual programming task. Appendix C contains a list of the common programming errors encountered in both simulation and systems programming development. Although the list is not comprehensive or exhaustive, it indicates many common mistakes.

Appendix D contains a set of guidelines that can help developers uncover the problems noted in Appendix C. Since the prototype model should contain all of the critical code, it serves as an excellent base for performing the verification. It is small, manageable, and inexpensive.

Tests should be written down in advance with the expected results clearly stated before a test is run. Remember that the purpose of testing is to find errors -- not to find ways that the program executes correctly. A test fails if no errors are found. All tests should be executed such that they can be replicated; interactive changing of variables during a run should be avoided.

9. CONCLUSION

We believe that simulation development has many parallels to both systems programming and operating system development. As such, a large number of concepts used by software engineers are directly applicable to simulation development. However, there are a large number of methods which are inappropriate. We have tried to identify those techniques which have been found useful and which ones are not. There is ample opportunity for researching the similarities that we have noted and for determining how current and future research in software engineering can apply to simulations.

APPENDIX A

The guidelines provided for simulation and systems programming are:

- Design from the perspective of the entities or objects flowing through the system to be modelled or developed.
- Identify what happens as the entity proceeds and what is the necessary control information used in deciding the flow.
- Identify clear boundaries between where things happen and do not happen in the model. For example, a part moving on a conveyor versus processing on a machine.
- Ensure that a clear interface and protocol exists for getting across each boundary and that the interface does not make any assumptions about the internal terrain of any area.
- Try to normalize and standardize the types of areas in the model so that the amount of unique logic will be minimized.

APPENDIX B

The following lists the standards used during development of the SMT model:

- Code should be 'modular' or 'black-box' and reflect the architecture.
- Code should be data driven from tables and any hardwired values should be avoided.
- Modules will not make assumptions about preceding and subsequent modules.
- Modules will have one exit and one entrance.
- Modules should be relatively short, preferably less than a page.
- Prefixes used with labels and variables to readily identify modules, data, etc.
- Use macros, equates, or synonyms (if supported) to provide meaning and to allow formulas to be specified in one location.
- Comments used for all non-trivial operations.
- Exceptions and key code to be highlighted.
- Indenting and alignment used to provide visual cues.
- Variables to serve one purpose and have one meaning.

APPENDIX C

- There is the off-by-one syndrome. This can occur in two ways. The index method might be origined at zero or one and not all places in the code account for this. The other way is for inequality testing to be coded incorrectly and a 'less-than' should have been a 'less-than-or-equal-to' condition.
- Index algorithms might not be accurate. Single vectors or two dimensional arrays can be segmented to reflect multi-dimension structures or the index method is computational. Is the table and slot being picked correct? Will the last or first element be picked?
- Iterative logic is not. A loop might not be executed depending on the context. A loop might be executed once and not a second time. A loop might execute twice. Or, a loop might loop a number of times. Simple errors in coding can cause some of these situations to fail and only by testing the zero, single, double, and triple iterations can the tester be assured that the code is working.
- Relations can be mis-coded. Besides contributing to the off-by-one, index, and iterative problems, the use of compound relations and negative logic can cause other problems. Code that should not be executed is executed and the reverse. Do you know that every line of code was executed in the order it was supposed to be?
- Errors and failure logic does not work. Programmers seem to have a tendency to test things that they know will work and as a result, the error and failure logic is never tried.
- Debris. Do things get discarded and cleaned up when they should? Do variables have values left after the first run that should have been reset? Debris or old values can contaminate the system in very subtle ways.
- The zero case. Can the code handle zero overhead entities flowing through the system? Can the code handle an entity being destroyed at various points in the code? The null case is very hard to program for and keep in mind at every point in the code.
- Deadlocks and critical races. Do Processes obtain more than resources? Can one entity pause in the system and another entity race by? Are two entities racing towards critical code that only one entity should be in at any one time? These are hard to identify and correct.

APPENDIX D

The following list provides some guidelines for designing test suites for simulation code:

- If possible single step and execute the prototype.
- If A and B are the boundaries, check for A, A+1, A-1, B, B-1, B-2.

- Set all values in the matrix to unique values and while single stepping verify value retrieval.
- Set up data to execute loops 0,1,2,3 times.
- Force all error code to be executed.
- Set up data to be deterministic and verify results with pen and paper (cycle times, etc.).
- If patterns of entities can occur, try: ABAB, AABB, AA, BB, BA, BABA, BAAB, ABBA.
- Schedule entities for close arrival and minimal delays (minimize flow).
- Schedule entities for long arrival times (one in system at a time).
- Run entities with zero processing, travel, setup times (also zero operators, etc.).
- If multiple runs are performed, look at variables for proper initialization.

BIBLIOGRAPHY

AutoSimulations (1986). AutoMod User Manual. AutoSimulations, Bountiful.

Balzer, R. (1985). A 15 year perspective on automatic programming. IEEE Transactions on Software Engineering, Vol. 11, No. 11.

Brooks Jr., F.P. (1975). The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading.

CACI (1983). SIMSCRIPT II.5 Programming Language. C.A.C.I., Los Angeles.

Chattergy, R. and Pooch, U.W. (April 1977). Integrated design and verification of simulation programs. IEEE Computer.

Clarke, L.A. (September 1976). A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering.

CMS Research (1985). MAST User Manual. CMS Research, Oshkosh.

Cohen, J.W., Fiore, A.M. and Larson, R.H. (1982). Structured modelling. Proceedings Winter Simulation Conference.

Dahl, D.J., Dijkstra, E.W. and Hoare, C.A.R. (1972). Structured Programming. Academic Press, London.

Duket, D. (1982). Implementation: a requirement for successful simulation. Proceedings Winter Simulation Conference.

Endres, A. (June 1975). An analysis of errors and their causes in system programs. IEEE Transactions on Software Engineering, Vol. 1, No. 2.

Freeman, P. (1975). Software System Principles: A Survey. Science Research Associates, Chicago.

Goldberg, A.T. (July 1986). Knowledge-based programming: a survey of program design and construction techniques. IEEE Transactions on Software Engineering, Vol. 12, No. 7.

Golden, D.G. (October 1985). Software engineering consideration for the design of simulation languages. Simulation.

Gomaa, H. and Scott, D.B.H. (1981). Prototyping as a tool in the specification of user requirements. Proceedings Winter Simulation Conference.

Hansen, P.B. (1973). Operating System Principles. Prentice-Hall, Englewood Cliffs.

Hayes, I.J. (January 1986). Specification directed module testing. IEEE Transactions on Software Engineering, Vol. 12, No. 1.

Hopcroft, J.E. and Ullman, J.D. (1979). Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading.

Jackson, M.A. (1975). Principles of Program Design. Academic Press, New York.

Kernighan, B.W. and Plauger, P.J. (1974). The Elements of Programming Style. McGraw-Hill, New York.

Knuth, D.E. (1968-1973). The Art of Computer Programming, Vol. 1-3. Addison-Wesley, Reading.

Miner, R.J. and Rolston, L.J. (1986). MAP/1 User's Manual, Version 3.0. Pritsker & Associates, West LaFayette.

Myers, G.J. (1978). Composite/Structured Design. Van Nostrand Reinhold, London.

Parnas, D.L. and Clements, P.C. (February 1986). A rational design process: how and why to fake it. IEEE Transactions on Software Engineering, Vol. 12, No. 2.

Pegden, C.D. (1985). Introduction to SIMAN. Systems Modelling Corporation, State College.

Pritsker, A.A.B. (1984). Introduction to Simulation and SLAM II. John Wiley & Sons, New York.

Ryan, K.T. (1979). Software engineering and simulation. Proceedings Winter Simulation Conference.

Schriber, T.J. (1974). Simulation Using GPSS. John Wiley, New York.

Shaw, A.C. (1974). The Logical Design of Operating Systems. Prentice-Hall, Englewood Cliffs.

Sheppard, S. (January 1983). Applying software engineering to simulation. Simulation.

Sheppard, S.B. et al. (1981). The effects of symbology and spatial arrangement on the comprehension of software specifications. Proceedings 5th International Conference on Software Engineering.

Weinberg, G.M. (1971). The Psychology of Computer Programming. Van Nostrand Reinhold, London.

Wolverton, R.W. (June 1974). The cost of developing large-scale software. IEEE Transactions on Computers.

Yau, S.S. and Tsai, J.J.P. (June 1986). A survey of software design techniques. IEEE Transactions on Software Engineering, Vol. 12, No. 6.

Yourdon, E. and Constantine, L.L. (1979). Structured Design. Prentice-Hall, Englewood Cliffs.

Zelkowitz, M.V., Shaw, A.C. and Gannon, J.D. (1979). Principles of Software Engineering and Design. Prentice-Hall, Englewood Cliffs.

AUTHORS' BIOGRAPHIES

KENNETH N. McKAY is Associate Director of WATMIMS, the University of Waterloo Management of Integrated Manufacturing Systems Research Group, and is responsible for the administration and planning of the group's activities. He has experience in R&D in the computer industry and has developed performance monitoring software, relational database software, and a variety of real-time systems for handling cheques in the financial industry. Mr. McKay's expertise includes: design evaluation, software engineering, software quality, systems architecture, and man-computer interfaces.

Kenneth N. McKay
Associate Director, WATMIMS Research Group
Department of Management Sciences
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1
(519) 888-4519

JOHN A. BUZACOTT is a Professor in the Department of Management Sciences at the University of Waterloo and is Director of the WATMIMS Research Group. His interests in manufacturing include the planning of integrated systems and the assessment of their performance, improvement of work flow, control of work in progress, and quality planning and analysis. Dr. Buzacott is also investigating methods for understanding the factors determining the impact of flexible manufacturing systems on industry. He is Departmental Editor in Manufacturing and Automated Production for the IIE Transactions, an Associate Editor for the Naval Research Logistics Quarterly, and on the Editorial Board of Queueing Systems, Theory and Applications of Material Flow. He has been President of CORS and Chairman of the NATO Advisory Panel on Advanced Study Institutes.

John A. Buzacott
Department of Management Sciences
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1
(519) 888-4009

JOHN B. MOORE is Associate Professor of Management Sciences and Associate Chairman of the Department of Management Sciences at the University of Waterloo. He is the designer of a state-of-the-art planning and scheduling system for microcomputers and is currently investigating the modelling of information and production flows in flexible manufacturing systems and the design of decision support systems for manufacturing. Dr. Moore's work also entails investigating the application of expert systems and the creation of effective strategies and tactics for large man-machine systems. He is particularly interested in man-machine interfaces utilizing graphics and animation.

John B. Moore
Department of Management Sciences
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1
(519) 888-4036

CHRISTOPHER J. STRANG, P.Eng., is a research assistant and graduate student in Management Sciences at the University of Waterloo. He received his B.A.Sc. in Civil Engineering in 1980 from the University of Waterloo. He has held various positions in industry with IBM, Ingersoll-Rand, and Hewlett-Packard before returning to graduate school and doing research for the University of Waterloo Management of Integrated Manufacturing Systems Research Group (WATMIMS).

Christopher J. Strang
Department of Management Sciences
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1
(519) 885-1211, ext. 3863