Simulation of IBM Enterprise System/9000 Models 820 and 900

by D. F. Ackerman
M. H. Decker
J. J. Gosselin
K. M. Lasko
M. P. Mullen
R. E. Rosa
E. V. Valera
B. Wile

The discovery and removal of logic design errors early in the development cycle is critical to timely availability of market-driven processor products. This paper describes the part played by simulation in the verification of the high-end models of the IBM Enterprise System/9000™ (ES/9000™) processor family, and how that effort advanced the state of the art of logic design simulation. The increased complexity of the ES/9000 design over that of the IBM Enterprise System/3090™ (ES/3090™) necessitated a larger simulation effort. New tools and methods were developed. Two simulation missions were established. Element simulation addressed ES/9000 functional elements (e.g., the storage controller) individually using the Compiled Enhanced Functional Simulator (CEFS), a software tool. System simulation tested two or more functional elements together using the Engineering Verification Engine (EVE), a special-purpose hardware parallel processor, and an attached IBM 3092 Processor Controller (PCE). The results achieved by simulation are discussed, together with the methods used and the impact these results had on the overall verification of the ES/9000 Models 820 and 900.

Introduction

It has been documented [1–3] that the cost of uncovering hardware design errors is high when working with custom VLSI circuits. Engineering changes are costly in terms of time and engineering effort. Hardware design errors often prevent complete testing of a function, which may delay the discovery of additional errors. All of these factors bear directly on the profitability of a product. The potential revenue and market advantage lost because of a long engineering test-floor effort is very large in today's increasingly competitive mainframe marketplace.

During the early planning stages of the IBM Enterprise System/9000™ (ES/9000™) processors, it was concluded that more errors had to be removed from the design *before* hardware was built. Simulation of logic design prior to its fabrication in hardware had played an important part in the verification of earlier machines. However, as then practiced, it would provide neither adequate nor expeditious verification of the ES/9000 machine architecture and its implementation. Its scope had to be increased. This paper discusses the part played by simulation in achieving this for the high-end processor models 820 and 900.

Two simulation missions were established: element simulation and system simulation. Each major functional element of the central processor complex, or CPC (central processor, storage controller, I/O subsystem, and

channels) would be simulated as a standalone entity in order to remove as many errors as possible. This was known as element simulation.

Once element simulation was completed, a full system simulation was then performed. System simulation was to execute in simulation the earliest tests performed on the engineering test floor after a machine was assembled and powered on. These tests consisted of power-on reset, certain manual operations, and the loading and execution of various operating systems, beginning with the systems assurance kernel (SAK), an internal IBM system exerciser. System simulation built system-level models combining the functional elements. It exercised these models on the Engineering Verification Engine (EVE) using SAK to discover system interaction design errors.

This paper is divided into two major sections discussing the two simulation missions, the simulation methods each used, and the testing performed. It concludes with a discussion of the results of simulation and their impact on the verification of the ES/9000 system.

Element simulation

Element simulation was responsible for verification at the board or card level of the following functional elements: central processor (CP), storage controller element (SCE), intercommunication element (ICE), and channel (CHE). The goal was the removal of at least 90% of the functional design errors. This number was chosen on the basis of the history of earlier simulation projects as well as the benefit of improved simulator performance and techniques. Furthermore, it was recognized early in the project that some areas of the design would be more efficiently verified at the system simulation level (e.g., multiprocessor design, I/O); thus, it would not be practical to find *all* errors at the element level.

Test design and definition

Once the hardware design had been reviewed for functional content and its availability date for simulation established, test design and definition began. A typical sequence for this part of the simulation process included the following:

- Deciding the source and type of stimuli needed for the test. In some cases, for example, a high activity level of a relatively random nature was required. This was provided by "behavioral macro" representations of interfacing logic external to the element under test. The macros were written in Basic Design Language for Cycle Simulation (BDL/CS) and built into the simulation model. They were driven by pseudorandom test vectors contained in control files. In other cases, unique and discrete events were most appropriate, and deterministic test cases were written.
- Deciding the environment in which the tests would be run. One key factor of the test environment was the model configuration. This could vary from a model with the maximum number of functional entities that could be interconnected to a model in which nearly half of the hardware was represented by functional macros. In the latter case, the computing resources required and the simulation run times were substantially less than in the former.
- Deciding the method of checking for correctness of function. This was closely related to individual test design, since it required knowing how the hardware design was supposed to work. As the number of individual tests increased, it became obvious that checking had to be done with no manual intervention. This meant that any test had to report pass or fail status, with the latter accompanied by supporting data of the test owner's choice. The self-checking test methodology also enhanced the regression-testing capability described later.

An important part of test design was the use of randomness in test cases and test drivers. This was an area of significant enhancement over past simulation efforts. Previously element simulation ended with the successful execution of manually written static tests. The random effect was built into several areas: the initialization of the model, the test itself, and the responses by other elements (e.g., behavioral macros) during the simulation. Tests were random in that no simple relationships existed between successive test vectors. However, the test vector sequence needed to be reproducible so that data could be collected when a design error was encountered, and to verify design fixes. For performance reasons simulation normally ran without full data collection. "Seeds" were used to initiate a given "random" series.

Environment

The Compiled Enhanced Functional Simulator (CEFS), an internal IBM software simulator running under Virtual Machine/Extended Architecture™ (VM/XA™), was chosen as the platform for element simulation. It offered high performance, flexible application interfaces, and very fast model compilation. The last was extremely important during the early stages of the design when changes occurred often.

The simulation was performed at a functional level, from a register transfer description of the design. The simulation models represented designs of 500 000 to 2 000 000 circuits. Typical test cases would run many thousands of logic cycles.

To limit the size of models and to increase performance, other elements in the system which interface with the element under test were implemented as behavioral macros

(i.e., scaffolded logic). For example, in SCE element simulation, the CPs were not included in the model, but rather CP macros, which behaved externally like a CP. This approach had several advantages:

- The macros could provide stress not normally achievable by the real element.
- The macros could be enhanced to provide logic not normally in the real element—for example, special recovery injection controls.
- Element simulation of one element was never gated by the design of another element.
- Macros could provide global system effects, such as multiprocessor support, so that these environments could be tested even in a single-element model.

Several different test drivers and test case languages were used in element simulation:

- Architectural verification programs (AVPs), which are manually written test sequences to verify a specific instruction (or combination of instructions) in a CP or I/O processor.
- Macros, as described above, used extensively as test drivers to provide random stimulus to the logic under test.
- Dynamic test language (DTL), which allows one to write low-level tests to alter any register or signal (facilities) in the model, clock the model, and expect certain values in facilities. This was useful in testing certain "non-mainline" functions, which were not easily testable by AVPs or macros.
- High-level languages, used to write programs interfacing with CEFS to set and extract facility values, clock the model, etc. These programs thus served as test drivers and performed complicated initialization and monitoring functions.
- SAK, an IBM internal operating system devoted exclusively to testing IBM large-system architectures, which provides random streams of instructions tailored to stress certain aspects of the Enterprise Systems Architecture/390™ (ESA/390™) architecture. It is described in more detail in the section on SAK-driven simulation.

System functions

All of the element-simulation efforts included testing system-wide functions. Historically, many of these functions were not tested until a real machine was built. They included the following:

Architected resets The operations performed by the processor controller element (PCE) for each of the architected resets [4] were emulated, and mainline tests were run afterward to verify that the element had reset properly.

Recovery During mainline tests, faults were injected into the model in either a controlled or pseudorandom manner. The recovery algorithms implemented by the PCE were emulated, and for recoverable errors, the tests were expected to complete as normal.

Degrade modes Many elements have portions of logic which can be disabled for recovery or test purposes. Examples include deleted lines in a cache, disabling the branch history table (BHT), and isolating, or "fencing," one element from another. These degrade modes were verified in element simulation.

Operator controls Examples of these include architected functions such as START, STOP, and Address Compare. Again, mainline tests were simulated as the function was exercised.

Measurements

To measure the progress and completeness of the simulation effort, several elements were monitored. Continuous tracking was done of the percentage of tests successfully completed on a line-item basis, as well as actual versus projected errors discovered. Key events in the model (e.g., queue-full conditions, interlocks) were monitored to make sure that the events of interest to designers were being stressed. Finally, simulation was periodically run in a special trace mode, whereby paths through the logic were tagged to discover those not being exercised. In a joint effort with the designers, this information was used to develop new tests.

Model checking/integrity

All element-simulation tests included automatic results checking, which provided operator-free testing and allowed extensive use of batch execution. This consisted of checking architected results and data integrity, and monitoring error checkers and hang detectors.

Results

Element simulation was extremely effective. The number of logic design errors detected ranged from about 87% in the CP to 95% in the SCE and ICE, and 97% in the CHE. Interface timing discrepancies, protocol errors, and differences of architecture interpretation were among the types of errors discovered. Since real Licensed Internal Code (LIC) was used, many errors were removed from it as well.

All elements realized improvements in random methodology and techniques for verifying system functions. This was the first extensive use of simulation for

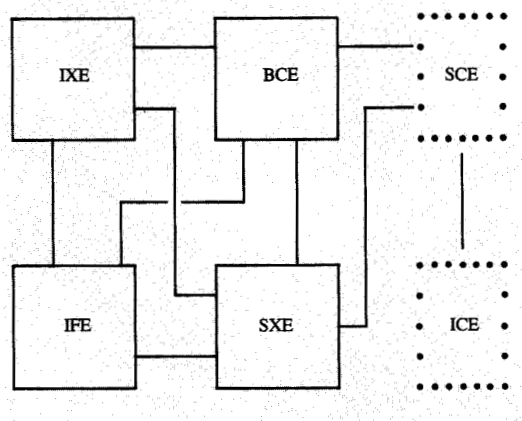

Figure 1

Structure of the central processor model. Boxes indicate hardware; dashed boxes indicate macros.

channels. It was also the first time that SAK was run in an element simulation environment.

Improvements to simulation tools were generalized to aid the entire CEFS simulation community. Among these were an event-driven cycle dump providing English phrases for specific events and functions that occurred in addition to model values, an event-driven error injector, and statistical analysis programs to determine function coverage and path analysis.

The following sections provide some more detail about the individual element simulation efforts.

- *CP element simulation*

The CP model consisted of the CP board comprising four TCMs: buffer control element (BCE), instruction fetch element (IFE), instruction execution element (IXE), and system execution element (SXE). To allow independent testing of the CP, a macro was created for the SCE. Macros were also created to provide signals for the timing facilities (including the external time reference), and to be able to handle some I/O requests and interrupts. The structure of the CP model is shown in **Figure 1**. The CP LIC was included as part of the model, and was exercised as needed to test the hardware design (verification of the LIC itself was the responsibility of the LIC development group).

Test drivers

To test at the instruction level, three major test vehicles were used: AVPs, random AVPs, and SAK. The test library of AVPs has been developed over many years, and had in the past been run on the engineering models of a new machine. This test library was enhanced for ESA/390 to include all of the new architecture items. Historically, the role of CP element simulation was completed with the successful execution of the AVP test library. Given the complexity of the ES/9000 CP, much more verification was needed.

Random AVPs are program-generated random streams of instructions with initial states and expected results. Because the expected results are generated by actually running the test stream on an ES/3090 machine, new ESA/390 architecture could not be included in the stream. These tests stressed machine design features as opposed to architecture, which is the domain of the normal AVPs and SAK. Some of the machine features of interest included

- Branch resolution and the branch history table (BHT).
- Out-of-sequence execution.
- Buffer control element (BCE) cache and dynamic lookaside table (DLAT) organization.
- General-purpose (GPR) and floating-point (FPR) register management.
- Storing into the instruction stream.

To test these features, program "generators" were written to produce random streams of instructions which were tailored for a particular scenario. For example, a branch test case could be generated which would have one or more of about 20 different types of branch loops, branch index tables, sequences of conditional branches, etc. These generators turned out to be extremely powerful.

All of the SAK drivers were run in CP element simulation except those stressing I/O and expanded store, since those areas were not completely modeled in the macros.

Stress environments

All of the types of tests mentioned above were simulated in environments which attempted to place additional stress on the model. These included the following:

- The BCE instruction and data caches were initialized randomly (or fully) with the instructions/data in the test. In addition, the data cache lines could be optionally loaded into the "synonym" classes of the cache. When the caches were preloaded in this manner, the timing effects of various cache states would appear very quickly. The DLATs had the same initialization capability.
- The BHT was preloaded either with random data or with known branch information from the test case.
- The GPR and FPR register management controls were randomly initialized so that different array positions were used to hold the architected registers each time a test was run.

- The rate at which the SCE responded to the BCE for fetches and stores was randomly varied, and the effects of slowing down or speeding up the SCE could easily be seen.
- The SCE macro was programmed to randomly send cache line invalidates, invalidates for page table entries, and storage key changes to the BCE, simulating multiprocessor effects.
- A program was written to inject I/O, external, or machine check interrupts at random points in the simulation of an instruction stream. The program would determine whether the interrupt should be taken, and if it should, would establish an interrupt handler to force the instruction address back into the test case so that the normal expected results would occur.

- *SCE element simulation*

The storage controller element (SCE) contains a high-speed buffer and the main storage arrays. It consists of data paths, arrays, and control logic. The SCE services data requests to and from the CPs, ICEs, and PCE, and must ensure that all data passed to other elements are not corrupted. The maintenance of data integrity was the main thrust of the SCE testing.

SCE element simulation was performed using two different CEFS test case drivers. The first comprised hard-coded test cases written in dynamic test language (DTL). The second was the use of a set of random drivers (macros) surrounding the SCE. This method was the primary test vehicle for the SCE.

Random SCE driver

The SCE interfaces with up to six CPs, one or two ICEs, and the PCE. All three can generate commands (e.g., store) to the SCE asynchronously. The random drivers and a control program were designed to send out randomly chosen commands at any time.

The drivers and control program had specific tasks. The basic responsibilities of the drivers were to send commands to be serviced by the SCE, to respond to commands passed on by the SCE, and to validate command responses and protocol. The largest tasks of the control program were to check data integrity on all interfaces and verify that the SCE adhered to architectural restrictions. The random drivers were built into the simulation model. The control program was a user-provided extension to CEFS.

If the drivers or control program detected unexpected results at any point during the run, the test case failed immediately. Debugging a failing test case was accomplished by using the control program output file. This file supplied a full account of all the commands, responses, and important SCE internal events that occurred during the failing test case. The output file also supplied specifics about the failure by explicitly stating which component (CP driver, ICE driver, PCE driver, or control program) flagged the error and what the error was. If the error was a data miscompare, the output file contained the actual versus the expected data. If the error was a response miscompare, the output file flagged the failing command and driver that detected the miscompare.

The CP, ICE, and PCE drivers used the following methods to choose commands. A different random number was generated each simulation cycle for each driver. The random number was then used to select a command to send across the interface to the SCE. A probability table for the commands, initialized at the start of the test case, was used to index the random number to a specific command. This initial probability table was varied from test case to test case. Once a command was picked, the driver decided whether or not the SCE could handle the selected command by monitoring the available lines on the SCE interface. If the interface indicated that the command could be sent, the driver began the sequence of sending the command. Otherwise, no command was sent to the SCE during that cycle. The probability tables also contained a no-op command, which, when chosen by the driver, sent no command across the interface on that cycle.

At the same time that the driver decided to send a command, a store or fetch address had to be requested if the command type indicated a memory access (there are control commands, such as SIGNAL operations, which do not access memory). In order for the memory controls to be verified, a small set of memory addresses was used by the drivers so that the probability of overlapping memory requests to the same address was increased. A set of 64 line addresses (128 bytes per line) was the maximum storage setup used. The line addresses were randomly picked at the start of each test case and were used throughout the test case. Whenever a driver requested a storage address, one of these lines was randomly chosen. Use of the limited address space allowed the test case to stress the exclusivity of the lines. At the same time, the small number of addresses ensured that if the SCE incorrectly stored data during a test case, a fetch command to that same address would likely follow, and the bad data would be flagged by the control program as the data crossed the interface back to the requesting driver. The random storage setup at initialization was performed by the control program. As the program picked addresses, it initialized the SCE main memory array with random data at the selected address and stored the same data in the program copy of the memory. Throughout the test case, the program would update its own copy of the address space whenever a driver sent a storage update in order to check data integrity. As a result, whenever a driver sent a storage fetch request, the SCE data were checked against

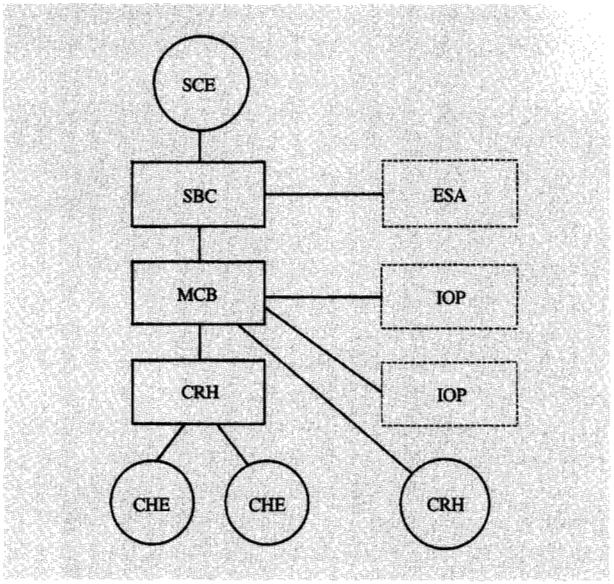

Figure 2

Structure of the intercommunication element. Boxes indicate hardware; circles indicate macros; dashed boxes indicate parts that could be either hardware or macros (depending on what was being tested).

the program copy of the data as the data returned to the requesting driver. Any miscompare caused the test case to fail.

After the SCE processed a driver command, a response was returned. It was the responsibility of the drivers to validate the SCE response. Unsuccessful responses were expected in many cases, such as when the requested address was currently locked out by another driver. Other examples of expected unsuccessful responses were when a driver, for testing purposes, generated an illegal sequence of commands, generated bad parity on an interface, or used an unconfigured address in its request. If the driver expected a different response from that which the SCE returned, an error was flagged and the test case failed.

Hang detects were implemented in the CP drivers to ensure that no commands were lost or overly delayed by the SCE. For a test case to be successful, every command sent by a driver had to be completed by the SCE within the hang detect interval.

A typical successful test case had about 150 requests from each CP driver, 150 requests from each ICE driver, and up to 10 PCE requests. Thus, a fully configured simulation model executed over 1200 SCE requests per test case. Over 200 000 000 random memory requests were sent and checked for proper execution. However, since these memory requests occurred in different SCE states because of the randomness of the test case, they exercised the SCE in different ways. As a result, thousands of memory states were tested.

- *ICE element simulation*

The intercommunication element (ICE) consists of four discrete interacting parts:

- Channel request handler (CRH)—a multiplexing switch for handling requests to and from the channels (CHE).
- System bus controller (SBC)—a crosspoint switch with an expanded storage controller (ESC) by which the ICE controls the expanded storage arrays (ESA), and communicates with the SCE and thereby with the CPs.
- Integrated off-load processor (IOP)—an interrupt-driven RISC processor for managing I/O operations.
- Multichannel buffer (MCB)—a multiplexing switch and buffer for communications between the IOPs, channels, CPs, and main and expanded storages.

Figure 2 illustrates the structure of the ICE and the communications paths between its parts.

Since each part had a character of its own, testing addressed both the individual characteristics and the interactions among parts. This dictated the use of macros to functionally stress the switching and buffer entities, and AVP test cases to test the IOP. Testing focused not only on the complete ICE but also on its individual parts and combinations.

IOP test

The IOP had to execute both architected and nonarchitected instructions. The latter were required to communicate with the CPs and the channels through the MCB. Tests addressed both of these functions. AVP test cases were used to test the IOP decode, execute, and put-away mechanism.

Single-instruction AVPs These manually written test cases were used to test architected instructions supported by the IOP and nonarchitected internal operations of the IOP. They were also used to test the accessing of main and expanded storages.

Random-instruction-stream AVPs A pseudorandom-instruction-stream generator was used for both architected and nonarchitected operations. These instruction streams were used to create interference between instructions of all types. The generator could vary the instruction streams according to specified parameters, including

- Instruction classes, e.g., branches, rotates.
- Inclusion or exclusion of specific operations.
- Types of interrupts allowed.

Judicious reselection of parameters resulted in discovering errors when the previous set of parameters no longer yielded new errors.

Switch/buffer test
This used the MCB, CRH, and SBC in a model with SCE, IOP, CHE, and ESA macros, with and without a macro for the second CRH. All macros (described below) except the ESA acted as drivers, generating requests to the real components in the model. The ESA was passive, servicing store and fetch requests directed to it.

MCB test The MCB interfaced with all other functional entities of the ICE. Testing it required generating requests from all possible sources in random order to stress the MCB.

CRH test Since the CRH could handle requests to and from as many as 64 channels, special attention was given to driving it with a variety of channel configurations and requests.

ICE macros
The macros used for ICE simulation were CHE, CRH, ESA, IOP, paging generator, and SCE. Several were replicated one or more times in a given model. The main rationale for the numerous macros was to force high coverage and high stress quickly and at a subelement level of the model, where debugging would be easiest. All but the ESA macro actively generated random requests under user control (biasing) to the design. Hundreds of runs were made, with many variations of the parameters of the driver macros. This detected 85–90% of the mainline function errors in the CRH/MCB/SBC complex. Checking for correctness of responses and for data integrity was built into all active macros. Audit trails were available for failure analysis.

The active macros shared many characteristics, so a detailed operating sequence is given for one of them (CRH), while a summary of function is given for the rest.

CRH random macro The CRH macro was used to represent one CRH of two attached to an MCB. It generated requests to the MCB (reads and writes). To cause contention in the MCB with requests from the other (real) CRH, the macro kept up to sixteen channels running at one time. All error checking done by the macro occurred at the time the MCB returned the command response to the CRH. Requests were one of three types: MCB buffer requests, requests to storage, and requests to the MCB channel communication area (CCA). CCA requests competed for some of the same MCB resources as the other two requests.

The generation of buffer and storage requests was as follows:

- Choose a random channel every cycle.
- If the channel selected is free and the store biasing permits, start a new buffer store.
- Finish the store data transfer to the MCB buffer.
- Send a storage command to the MCB; when the response returns, mark the store done and ready to be fetched.
- Use a second random channel to point to the "store done" status of another channel. If data transfer to storage is complete, and if data are not being fetched by any other channel, send a fetch storage command to the MCB.
- When the fetch data return, compare them to the data originally stored, log any errors, and mark both channels (fetch and store) free for further use.

Channel (CHE) random macro The CHE macro stressed the CRH by loading it with as many as 64 channels in differing configurations. The macro was a random-biased command generator representing sets of four to eight channels attached to the CRH. This ensured contention across the bidirectional bus interface with the CRH.

IOP random macro Only about 10% of all IOP instructions resulted in communication with the MCB. In practice, the traffic presented to the MCB at the IOP interface was quite low, and required additional communication with the CPs and the channels through LIC. The IOP macro was designed to simplify and maximize the interaction between IOP and MCB. It generated requests to the MCB for fetch/store activity to main and expanded storages, and for CCA operations in the MCB. It also could respond to interrupts from the CPs that were forwarded by the SCE macro.

SCE macro This macro responded primarily to requests for fetch/store activity from/to main storage. It had the added ability to present CP requests for work (interrupts) to the IOP. These were generated at user-controlled intervals, and checking was done for the IOP response to each request.

Paging random macro Interfacing with the SBC through the SCE macro, the pager requested page transfers to and from the expanded storage. Besides its use as a page generator, this macro was useful in creating contention for SBC resources during random runs.

ESA macro This representation of the expanded storage arrays received and processed requests from ICE. It was passive in that it did not initiate requests as did all the other macros.

• *Channel simulation*

The channel is an engine providing a data path from the CPC to external devices such as control units (e.g., 3990 storage control), other CPCs [via channel-to-channel (CTC) connection], or IBM internal test devices.

Channel simulation tested the System/390® parallel and high-speed fiber optic channels; included in this activity were connecting different I/O devices (some modeled, some as macros to the channel), and providing support for other sites to integrate the channel into their simulation environment.

Environments

The base configuration for simulation was a modeled channel connected to a subset of the ICE, functional macros for the IOP, CP, SCE with main storage, and a macro for the test device (universal programmable device emulator). The channel used real LIC, previously tested by its developers, to provide functional operation. A variety of channel diagnostics and specialized internal code were also written.

Additional configurations utilized the base simulation model augmented to include actual physical device hardware [e.g., connection converters (9032, 9033), directors (9034, 9035)], multiple test devices per channel, multiple channels, and/or a CTC connection. Another alternate model configuration included physical IOP hardware, LIC, a logic support station (LSS), and a logic support element (LSE) macro to provide a platform for testing IOP LIC, subsystem resets, and recovery.

Test drivers

The general simulation philosophy included generated random, self-checking, and seed-based (for reproducibility) test cases. The main testing strategy was based on the architected I/O instruction level utilizing the LIC for functional operation. The random I/O tester (RIOT) environment was established to fulfill this objective. RIOT was designed to provide a level of test comparable to the test-floor SAK procedures for testing the channels. RIOT generates random test cases for each channel and device consisting of random architected I/O instructions (Start, Clear, Halt, Resume, Cancel subchannel, etc.) for each device. Then random channel command word (CCW) programs are generated for each start subchannel instruction. RIOT was initially designed to work with the internal test device. When support was extended to include other devices, it became apparent that more CCW control was necessary. RIOT I/O device extension routines (RIDER) was developed to accommodate restrictions on CCW programs for any external device. RIDER provided flexibility in tailoring CCW programs for any user-defined device by constraining the CCW programs and status responses. RIOT also did random configuration of the device emulators by randomizing several aspects such as speed, type (byte or block for parallel), and frame sizes (for fiber optic channels). RIOT was also responsible for results checking, which included data integrity, architected responses, and responses to the IOP.

Multiple site support

Since the channels connect to a wide variety of external devices or control units, and to systems other than the ES/9000, channel simulation supported multiple sites in connecting the channel in their environment. This included incorporating their hardware devices into the model (connection converter, connection director, test devices, etc.) or helping to include the channel in their simulation by providing a mechanism (RIOT/RIDER) to govern the random I/O instruction generation to specific devices, since they may require device-specific CCW chains (3990 storage control, CTC, etc.). This required compatibility across operating systems (MVS and VM), support of environments using the EVE simulator, and support of other cross-simulator efforts.

System simulation

After each element had been simulated in the element environment, system simulation combined the individual elements into full-system models that were run on the Engineering Verification Engine (EVE) system.

The overall goal of simulation was to reduce significantly the time required by the engineering test floor to power-on a real machine, reset that machine to a good running state, then IPL an operating system (SAK followed by MVS and VM) successfully. The challenge of system simulation was to provide a machine to engineering test that could

- Successfully execute a power-on reset (POR).
- Perform initial LIC load.
- Load the SAK operating system.
- Run various SAK test case drivers error-free for at least three minutes each.

An additional goal was to verify the tools needed by the test floor to debug the errors it encountered.

System simulation was to remove 8% (over and above the 90% removed by element simulation) of the total hardware design errors prior to hardware being built. This would leave 2% of the total to be removed on the test floor. System simulation utilized the EVE hardware simulator as its primary test vehicle.

EVE is a custom parallel computer that executes models compiled from hardware descriptions written in BDL/CS [5]. EVE is channel-attached as a device to an IBM mainframe "host" running VM/XA. The main function of the host is to control and interact with simulation "jobs" executing on EVE. The main advantage of EVE over CEFS is higher

cycle throughput, and this advantage increases as the size of a model increases.

Early in the planning stages of the ES/9000 machines, it was decided to divide the system simulation effort into two distinct groups. The first group, known as Mainline Test, utilized SAK test cases to verify the logical function of the machine in a normal machine operating environment. This test effort included architectural verification of the CP and I/O functions as well as the interaction between them in the storage hierarchy. The second group, known as Resets/Recovery Test, was responsible for testing the reset functions of the machine (POR, IPL, and other types of machine resets), error recovery, and other related manual operations—all from a system perspective. This was accomplished by attaching an actual processor controller element (PCE) to the EVE simulator to exercise the system model with the actual processor controller LIC (PCLIC) through a pseudo-hardware interface.

SAK-driven simulation

SAK consists of a control program and a set of test routines designed to test all aspects of the machine architecture. The control program provides the basic operating environment, and the test routines perform all test case creation, expected results generation, and results checking. On the test floor, the control program, test routines, and test cases all execute on the system under test. For simulation, a test case delivery mechanism was required to allow the control and test routine overhead to be offloaded to a host system. The SAK-driven simulation (SDS) system was developed to provide this capability and fully exploit the high-performance EVE simulator. By transferring the overhead of SAK to a host system and executing the test cases on an EVE model, this system was able to achieve a test case throughput rate equal to about one thousandth of that attained by SAK running on the ES/9000 test-floor system.

In the SDS system, the control program and the test routines run on a VM/XA host system.* Only the test cases and a minisupervisor function run on the simulation model. The minisupervisor dispatches test cases from a work queue, processes test interrupts, and controls when tests are finished and new queues of work should be loaded. A host-resident mapper program is used to load new test case queues into, and transfer actual test results out of, the memory of the simulated model. A queuing mechanism was designed to attain high EVE efficiency. This mechanism allowed many test cases to be grouped together into long work queues for dispatch to the model, which greatly improved test case throughput by minimizing the number of host–model interactions. Finally, a record and playback function was added to guarantee the reproducibility of simulation failures and aid in isolating them.

The SDS system was originally put into practice by the ES/3090 Model S system simulation group. Additional advances were made during the ES/9000 simulation effort to improve usability and performance. These improvements included an automatic job submission and control system, an automatic error isolation and trace process (described below), an enhanced mapper program yielding wider applicability of function and improved performance, and the development of an associative main memory scheme allowing large address space tests to be run on a hardware simulator with a fixed-memory modeling capability.

Debugging techniques

Aggressive verification schedules coupled with the number of errors projected to be found during system simulation made it essential to develop techniques to reduce the time required to isolate and debug errors. Early experience with SDS had shown that it was very difficult and unproductive to isolate and debug simulation failures because of the limited model trace capability of EVE. The CEFS software simulator, although much slower than its hardware counterpart, possesses a very powerful trace facility. The solution was to couple the EVE and CEFS simulators into a complete automated simulation process:

- The user submits a SAK test run to the EVE simulation system.
- SAK test cases are simulated on the model while SAK performs test creation, results generation, and checking of prior tests.
- Test case progression is tracked through the system and all SAK-model stimuli are recorded.
- At user-selected intervals (because of the large volume of data) the state of the model is saved. This saved model state is called a checkpoint.
- A model error is detected by SAK.
- The control software isolates the model failure to the last recorded checkpoint preceding the load of the failing test case.
- The model is reset to this model state.
- The recorded stimuli are "replayed" to the model up to the start of the failing test case.
- At this point a new checkpoint is obtained and simulation is terminated.
- This new model state and the recorded stimuli are then transported to CEFS.
- A complete all-events trace is obtained on CEFS and returned to the user for analysis.

*The VM/XA operating system was required so that the capabilities of the simulated model could be matched in the test environment and exploited by the test routines. For example, by defining multiple virtual central processors on VM/XA, SAK was able to create multiprocessor and extended architecture test cases for the simulated model while running on a uniprocessor host.

In practice, this process proved to be superior to previous techniques used on EVE, and reduced the time to isolate and debug a model failure from days to *hours*. The improved problem-turnaround time lessened the discovery of duplicate errors, thereby freeing up valuable human and simulation resources.

Processor controller–EVE attachment
Early debugging of the error recovery and reset functions in simulation required that the PCE be "attached" to the simulated machine model. This was accomplished by physically connecting an actual PCE to the EVE. This involved changes to the PCLIC, the development of a PCE-to-EVE adapter card, special internal code in EVE, and extensions to the simulated model. A record and playback mechanism was added to guarantee error repeatability and aid in error isolation. The PCE–EVE system could also be used with the SDS system. This provided in system simulation the environment of the engineering test floor, wherein PCE function testing is performed while SAK runs on the actual CPC hardware.

- *Mainline test*

System architectural test
Mainline test was split into two groups, one responsible for CP and SCE test and the other for I/O testing. Each group defined which model configurations (number of CPs, SCEs, channels, etc.) it would require to verify their specific portion of the machine. The CP/SCE test group did not require an I/O subsystem (IOSS) in their test configurations; to create contention for data and hardware resources they wrote macro behaviorals in BDL/CS to drive the IOSS interface. The I/O test group required a full system model (CP, SCE, and IOSS without a PCE), and represented actual devices (printers, terminals, DASD, etc.) with macros. Each group utilized SAK as its primary test case driver to verify the machine operation, and each was responsible for modifying the SAK user parameters to optimize test efficiency and uncover errors as quickly as possible. Each group experienced difficulty during the early test cycle in debugging errors on EVE in a timely fashion because of a large quantity of duplicate errors, which were difficult to mask, and contention for the EVE resources. Many debug enhancements described previously (especially the EVE-to-CEFS playback capability) were solutions to these problems.

The experience during system simulation confirmed that the use of SAK in simulation was efficient in the removal of errors from the design.

"Undefined" test
System simulation also took on responsibility for testing the logic for propagation of undefined states after the reset of a machine. Simply, a value of "undefined" was placed in each bit of every latch in the machine and simulation was run for 100 cycles. The hardware reset bit in the simulation model would be activated and an additional 100 cycles run. The latches would be examined to ensure that no "undefined" values were present, indicating that the reset was correctly executed. This effort removed errors that had previously been removed *only* on the test floor, requiring new hardware to be built.

Performance measurement
Machine throughput (performance) projections were accomplished by using instruction-trace-driven simulation models [6]. Once a functioning machine was available, actual throughput measurements were made by executing the tests on real machines. This activity also verified the accuracy of the trace-driven models. For the ES/9000 machines, as a complement to these steps, the throughput measurements were run against the hardware simulation models. This allowed not only more timely verification of the trace-driven models, but also earlier identification and removal of performance problems.

Regression test
Once the first hardware was built, a set of regression tests, including performance tests, was executed in simulation to verify machine engineering changes. This proved successful by identifying unsuccessful fixes or errors introduced by new functions, and providing a level of confidence in the physical hardware construction.

- *Resets, recovery, manual operations, self-test*

Resets, recovery, and manual operation tests were run on an EVE system model using real PCLIC executing on the attached PCE. The types of error found included hardware design errors, LIC errors, errors in the PCLIC algorithms controlling resets and recovery, and errors in the hardware-dependent data tables (design data) used by the resets and recovery PCLIC. This testing was not intended to find all of the errors in the PCLIC (another test organization was responsible for that), but some were found through its use. Some of the resets and recovery support functions were tested by using the CEFS simulator with a tool to verify the design data comprising the Q-bus data table (QDT). The Q-bus is an interface between the PCE and the CPC that can be used without stopping the CPC clocks; it is the primary means for the PCE to control the reset and recovery actions in the CPC. Self-test was not tested in this way. Instead, it used an EVE model built from two different design languages, as described below.

Resets
Resets simulation tested the nonerror paths of the reset algorithms corresponding to the architected functions

tested, and, additionally, some of the error paths due to real logic errors. The following functions were tested: power-on reset (POR), system reset, system reset clear, initial program load (IPL), IPL clear, CPU reset via SIGP, and initial CPU reset via SIGP. The PCE–EVE environment was used for these tests; the resets were executed using the real PCLIC. For the SIGP resets tested, the instructions were placed in the system area of main storage of the model, and a restart was issued from the PCE.

In general, two methods were used to verify the resets testing. First, the error data read by the PCE were used to ensure that no error checkers came on during the reset. These data would be available on the test floor in the error logs on the PCE; in simulation they were directly examined in the EVE output files. Second, the reset checkpoint was saved and given to Mainline Test, where it was used as an initial state for a SAK simulation run.

Toward the goal of executing an IPL on an operating system on the test floor, the steps of a SAK IPL were analyzed, and corresponding simulation tests defined and executed. After POR, the next step was to IPL the SAK minisupervisor program mentioned earlier. The simulation model contained a macro that modeled the IPL I/O device. This macro contained the 24 bytes of data to be read in by the initial channel command word (CCW) of the IPL. These data contained the minisupervisor initial PSW, and the CCW to read in the minisupervisor data. After the minisupervisor was loaded, the CP was started from the PCE, and the state of the model saved. This checkpoint was then used as a starting state for a SAK simulation run to verify the IPL. Those steps not covered by this minisupervisor IPL were covered by mainline tests using SAK programs that performed the I/O operations and executed operations specifically used by the real SAK supervisor on the test floor.

Recovery

Recovery simulation executed the recovery algorithms implemented in PCLIC. In addition, errors were injected when the model was in different states to ensure that general recovery would work under different conditions (e.g., will CP recovery succeed if an error is injected during an I/O operation?). Simulation runs were started from an initial model state provided by the model bring-up team. This model state was close to the POR state. However, since PCLIC is not part of the model, a POR would have to be executed to get its state to match that of the model—a step of several hours. As a shortcut, code was written to initialize fields in the configuration tables used by the PCLIC, so that a full POR did not have to precede a recovery test run.

SAK was used to generate activity in the model, although some handwritten loops were also used. Intermittent faults were injected at random or at some particular model event. This was done from the simulation host instead of the PCE for efficiency and better conditions control. Since element simulation of recovery had been done, system simulation concentrated on errors such as interface errors and recovery scenarios that involved multiple elements (e.g., CP or SCE errors during execution of I/O or page operations).

Successful completion of the SAK test cases indicated successful error recovery. For errors that invoked general recovery, several errors were injected and recovered sequentially. For additional verification, the same types of data available on the PCE and EVE for problem debug were inspected.

Manual operations

The environment built to test resets and recovery lent itself to testing some of the PCE manual operations. Several manual operations were tested, including alter/display, address compare, and some of the engineering debug commands.

QDT verification

Most of the PCE–CPC communication is done by reading and writing Q-bus registers, which are defined to the PCLIC in the QDT. To test the validity of these hardware-dependent data, a tool was built with which the Q-bus read/write activity represented by each QDT entry was simulated and verified.

Self-test

The self-test function is a stuck-fault test provided for manufacturing and on-line hardware diagnostics. Using a seed, the self-test hardware generates pseudorandom data, applies a clocking sequence to produce data flow through the hardware, and compresses the result into a signature register. This testing required a model containing components built with both a gate-level language [Basic Design Language for Structure (BDL/S)] for simulation of low-level clocking and scanning functions, and a register transfer-level language (BDL/CS) for simulation of the self-test supporting elements. To accomplish this, mixed-language simulation (MLS) was applied. Using MLS, errors were found, particularly in the clocking and scanning design. Removing these errors in simulation avoided serious test-floor consequences for two reasons: 1) clocking and scanning errors are much more difficult to debug on the test floor than are other types of errors; 2) scanning is frequently used to gather data to debug other errors.

Conclusion

Simulation detected 97% of the hardware design errors discovered during the ES/9000 high-end verification. For

each hardware design error that escaped to the test floor, extensive escape analysis was performed to determine why simulation did not find it during the various stages of testing. Many of these errors could have been found in simulation if the proper instruction sequence had been executed with the machine in a particular state. These escapes are attributed to the pseudorandomness of the SAK test cases in conjunction with the relatively short nature of the simulation test cases. These errors typically are a manifestation of the machine construction rather than an architectural design violation, and are found after minutes or hours of running on an actual machine. These findings, along with designer areas of concern, have been used to improve the testing done in simulation, where feasible. Additionally, they have been forwarded to the SAK development team to enhance their test case functional coverage, and to tools support groups for more powerful simulation tools.

Simulation had a marked impact on the verification of the ES/9000 high-end machines. Power-on reset ran successfully once physical and assembly defects were removed. From machine assembly to the SAK operating system actually running on the ES/9000 took *31 days*—a major success when compared with previous machine experience. The contribution to this achievement made by simulation validated the strategy of executing in simulation those test floor bring-up procedures whose success or failure gated further test floor efforts. The test floor now had a machine on which basic bring-up and debugging tools worked—*and had it earlier than ever before.*

Appendix: Glossary

The following definitions are provided for the acronyms found in this paper:

- AVP Architectural verification program. A manually written test case for specific machine instructions.
- BCE Buffer control element. Part of the CP which contains the cache.
- BDL/CS Basic Design Language for Cycle Simulation. A language used for hardware design.
- BDL/S Basic Design Language for Structure. A language used for hardware design.
- BHT Branch history table. Part of the CP which records previously executed branch instructions.
- CCW Channel command word. An instruction executed by a channel.
- CEFS Compiled Enhanced Functional Simulator. A simulator implemented in software.
- CHE Channel element. Provides a path from the CPC to external devices.
- CP Central processor. The component of the CPC that fetches and executes instructions.
- CPC Central processor complex. The total of the hardware of a system, excluding the I/O devices.
- CRH Channel request handler. Part of the ICE. A multiplexing switch for handling requests to and from the channels.
- CTC Channel-to-channel. Between channels.
- DASD Direct access storage device. Large-capacity storage media, usually disks, capable of being randomly accessed.
- DLAT Dynamic lookaside table. Part of the BCE. Used for virtual-to-real address translation.
- DTL Dynamic test language. A low-level test case language.
- ESA Expanded storage arrays. Part of the ICE.
- ESC Expanded storage controller. Part of the ICE.
- EVE Engineering Verification Engine. Hardware parallel processor built for simulation.
- FPR Floating-point register.
- GPR General-purpose register.
- ICE Intercommunication element. The component of the CPC that handles channel and expanded storage requests.
- IFE Instruction fetch element. Part of the CP which decodes instructions and prepares them for execution.
- IOP Integrated off-load processor. Part of the ICE. An interrupt-driven RISC processor for managing I/O operations.
- IOSS I/O subsystem. The control units and devices attached to a CPC.
- IPL Initial program load. The loading of a program from an external device, and the initialization of that program.
- IXE Instruction execution element. Part of the CP which generates operand and instruction addresses and executes most of the architected instructions.
- LIC Licensed Internal Code. Instructions executing internal to the CPC. Licensed Internal Code is copyrighted and is provided to the customer under the terms and conditions of the applicable written agreement between the customer and IBM.
- LSE Logic support element. Provides a communication path from the PCE to an LSS.
- LSS Logic support station. Allows a component of the CPC to communicate with the PCE.
- MCB Multichannel buffer. Part of the ICE. A multiplexing switch and buffer for communications between the IOPs, channels, CPs, and main and expanded storage.
- MLS Mixed-language simulation. Simulation using models built with more than one design language.
- PCE Processor controller element. The component of the CPC that provides operator interface, support, recovery, and diagnostics.

- PCLIC Processor controller LIC. LIC executing on the PCE.
- POR Power-on reset. The architected reset performed as part of the power-on sequence.
- PSW Program status word. A hardware facility containing information used by a CP to control instruction sequencing.
- QDT Q-bus data table. A compilation of symbolic addresses used by the PCE to access the CPC.
- RIDER RIOT I/O device extension routine. Allows RIOT to accommodate restrictions on CCW programs for any external device.
- RIOT Random I/O tester. Generates random test cases used in channel simulation.
- SAK Systems assurance kernel. An internal IBM system exerciser.
- SBC System bus controller. Part of the ICE. A crosspoint switch with an ESC by which the ICE controls the ESA and communicates with the SCE and thereby with the CPs.
- SCE Storage controller element. The component of the CPC that services storage data requests.
- SDS SAK-driven simulation. Simulation using SAK as the test program executing on the model.
- SIGP Signal processor. An architected instruction by which CPs communicate control information to each other.
- SXE System execution element. Part of the CP which executes complex system instructions, e.g., I/O instructions.

Acknowledgments

Many of the random-testing concepts which were key to the success of element simulation were the work of G. D. Huber, B. Chu, M. Billeci, J. E. Roets, K. M. Pandey, J. H. Klaus, and M. H. Decker. D. E. Lackey is recognized for the planning of the system simulation SAK–EVE test methodology. The SAK system is based upon early work by J. Lum and is today supported by many people, including M. F. Sassaman, J. Strong, J. R. Birtles, D. M. Balog, and T. J. Bohizic. The SAK-driven simulation system was developed as a joint effort among G. R. Deibert, D. F. Ackerman, S. S. Chu, D. C. Hughes, and R. G. Sheldon of the simulation tools group and M. F. Sassaman, J. Strong, J. R. Birtles, and J. R. Larew of the SAK group. G. J. Tuma and G. W. Gorman are recognized for their efforts in debugging and bringing the initial SAK-driven simulation system into production during the ES/3090 Model S development effort. The PCE–EVE system was developed by D. F. Ackerman, T. G. Fike, K. S. Schramm, and M. J. Gyves. It was an extension of earlier work done by J. D. Eggleston, S. L. Kessler, and M. Sing. W. R. Bissinger, G. R. Deibert, and C. Warburton are recognized for their efforts on mixed-language simulation. The authors thank R. G. Sheldon and C. S. Kischuk for reviewing this paper and making numerous suggestions for improvement, and C. A. Logan for suggesting, encouraging, and supporting its writing.

References

1. Michael Monachino, "Design Verification System for Large-Scale LSI Designs," *IBM J. Res. Develop.* **26**, 89–99 (1982).
2. R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process," *IBM J. Res. Develop.* **26**, 12–21 (1982).
3. S. G. Tucker, "The IBM 3090 System: An Overview," *IBM Syst. J.* **25**, 4–19 (1986).
4. *ESA/390 Principles of Operation*, Order No. SA22-7201-00, March 1992; available through IBM branch offices.
5. D. K. Beece, G. R. Deibert, G. P. Papp, and G. F. Villanti, "The IBM Engineering Verification Engine," *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988, pp. 218–224.
6. P. G. Emma, J. W. Knight, J. H. Pomerene, T. R. Puzak, and R. N. Rechtschaffen, "Simulation and Analysis of a Pipeline Processor," *21st Winter Simulation Conference Proceedings*, IEEE Press (Catalog No. 89CH2770 9), 1989, p. 1047.

Received April 18, 1991; accepted for publication November 27, 1991

Dennis F. Ackerman *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (DFA at PK705VMA).* Mr. Ackerman joined IBM in 1982; he is currently an Advisory Engineer in Simulation Subsystem Development. He has worked on the development of the EVE simulation system since its inception. His principal activities and interests are in the areas of simulation hardware accelerators and simulation test methods. Mr. Ackerman received his B.S. in electrical engineering from the University of Colorado in 1982, and his M.S. in computer engineering from Syracuse University in 1989. He is a member of Eta Kappa Nu and Tau Beta Pi.

Mark H. Decker *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (MDECKER at PK705VMA).* Mr. Decker joined IBM in 1985 in Channel Simulation. He received a B.S. in electrical engineering from Carnegie Mellon University in 1985 and an M.S. in computer engineering from Syracuse University in 1987. Mr. Decker is a Development

Engineer, and manages the SAK I/O Characterization Department. He is a member of IEEE, Eta Kappa Nu, and Tau Beta Pi.

Joseph J. Gosselin *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (GOSSELIN at PK705VMA).* Mr. Gosselin received a B.S. in physics from Boston College, joining IBM in 1968. Through 1975 he held several programming positions in software performance and MVS operating system support. In 1976 he joined Large Processor Engineering, where he developed LIC for the 3036 Processor Controller. From 1983 through 1987 he was manager of LIC and diagnostics development for the 3092 Processor Controller. In 1987 Mr. Gosselin joined Processor Simulation, where he is an Advisory Programmer.

Kevin M. Lasko *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (LASKO at PK705VMA).* Mr. Lasko is currently an Advisory Engineer in Processor Simulation. He joined IBM in 1978 as a Junior Engineer working in the Bond, Assembly, and Test function. Since 1981 he has worked on the simulation of the ES/3090 Storage Controller and has been involved with the ES/9000 simulation effort since its inception. His interests include storage hierarchies and personal computers. Mr. Lasko received his B.S. in electrical engineering from Union College in 1978, and his M.S. in computer engineering from Syracuse University in 1983.

Michael P. Mullen *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (MULLEN at PK705VMG).* Mr. Mullen joined IBM in 1976; he is a Senior Programmer. He worked in Processor Controller development prior to joining Processor Simulation in 1986. Mr. Mullen received a B.S. in computer science from Union College in 1977, and an M.S. in computer/information sciences from Syracuse University in 1981.

Ruth E. Rosa *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (ROSA at PKSMRVM).* Ms. Rosa joined IBM in 1986 in Processor Simulation. She has worked on the simulation of the ES/9000 system recovery functions, principally in the CP and Vector areas. Ms. Rosa is a Staff Engineer in Product Design. She received her B.S. in electrical engineering from Ohio State University in 1985.

Ernest V. Valera *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (retired).* Dr. Valera joined IBM in 1961. He worked in Scientific Computation, Telesatcom, Engineering Design Systems, and ES/3090 System Controller Design before joining Processor Simulation as an Advisory Engineer in 1984. He received a B.S. in electrical engineering from the University of Detroit in 1961, an M.S. in management engineering from Rensselaer Polytechnic Institute in 1964, and a Ph.D. in business administration from the New York University Graduate School of Business in 1973. He is a member of Tau Beta Pi. Dr. Valera retired from IBM in 1991.

Bruce Wile *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (WILE at PK705VMG).* Mr. Wile joined IBM in 1985 and is a Staff Engineer in Processor Simulation. He participated in the planning and implementation of the SCE random simulator. Mr. Wile received a B.S. in computer science from Pennsylvania State University in 1984.