Designer-level verification using TIMEDIAG/GENRAND

by B. Wile

TIMEDIAG/GENRAND is a tool set used on various portions of the CMOS processor for the IBM S/390® Parallel Enterprise Server Generation 4 to assist in designer-level logic verification. The concept of surrounding the logic design (hereafter referred to simply as "logic") under test with irritator behaviorals, a methodology developed and proven effective on larger simulation models, is moved to the designer level without the overhead of writing multiple behaviorals. Rather than writing source-level (e.g., VHDL, C code, etc.) behaviorals, the method creates an external stimulus to the design by using a series of generalized timing diagrams that obey the interface protocols of the logic under test. These timing diagrams are entered using the TIMEDIAG (timing diagram) editor. The effort required for logic verification is thus limited to understanding and laying out the interfaces to the design—a task that must be done for any well-designed unit of logic, regardless of whether or not it is being verified at the designer level. Once the timing diagrams are written, GENRAND (general random driver) is invoked to run simulation on the design. GENRAND randomly initiates the timing diagrams that obey the interface protocol, causing many different input and output permutations. This simulation is very effective in testing the logic implementation.

Introduction

The burden of verifying a single designer's logic has often fallen upon the individual designer. While substantial focus is put on testing larger portions of a logic design (e.g., an entire processor), it is often more efficient to remove logic design defects earlier in the process. But in order to verify many single functional logic macros within a design, a tool set must be supplied to the logic designers that allows for easy simulation of one's own design while maximizing the coverage.

In this paper, the TIMEDIAG/GENRAND tool set, developed to assist in designer-level logic verification, is described. The models on which the tool set is based are discussed, along with the requirements for solving the problems of designer-level verification. The features of TIMEDIAG/GENRAND are detailed, followed by a representative application used during implementation of the S/390* G4 system.

- *The irritator behavioral/test-case driver model*

A classic method of verifying logic is the use of irritator behaviorals. In this model, the logic under test is "surrounded" by dummy logic whose purpose is to drive the inputs and check the outputs of the logic under test. Because this dummy logic, or "behavioral," is used solely for logic verification, the choice of source language is not restricted to the design language. In later stages of verification, behaviorals are replaced with the real design that was being emulated. But by using behaviorals first at a lower level of simulation, a verification expert can exert greater control over the smaller piece of logic, thereby

0018-8646/97/$5.00 © 1997 IBM

testing more permutations and states in less time. Using irritator behaviorals is a highly effective method for verification [1].

A "behavioral" generally consists of two parts that work hand in hand: the interface protocol handler and the driver. The interface protocol handler is prompted by the driver to provide a specific stimulus to the logic under test. When the logic responds to the stimulus, the interface protocol handler receives the response and packages it back to the driver.

The driver can be thought of as the "brains" of the behavioral. It decides what to send into the logic and verifies the correctness of the output. A typical driver uses a probability table to choose which legal stimulus it will initiate. The checking logic that must be built into the driver can be far trickier to implement. Because the complexity of the logic under test generally increases with its size, a driver's checking logic on a large model is often quite complex. This is less of a problem with smaller, designer-level models, where fewer permutations and less complexity make for simpler prediction of outputs. However, it is a key to designer simulation, because a good methodology will be able to capitalize on the simpler checking requirements.

The interface protocol handler is the interpreter between the driver portion of the behavioral and the logic under test. With regard to the logic under test, the behavioral emulates the interface actions of the real logic that will surround the logic under test. When the driver portion of the behavioral "decides" to send a sequence, it is up to the interface protocol handler to create the correct signals for the logic. The interface protocol handler also monitors the outputs of the logic, packaging output sequences back to the driver so that it can check for correctness.

- *The challenges of designer-level verification*

The challenge of applying the irritator behavioral model to designer-level logic verification is less a technical challenge than a time-to-market concern. A quality irritator behavioral model is quite effective, but takes time to plan and implement. While it is a cost-effective approach to verifying large portions of logic, writing individual irritator behavioral models for testing each designer's logic is not feasible.

Without the power of irritator behaviorals, designers traditionally test their logic with a series of hard-coded test cases. But writing hard-coded test cases to verify one's own logic creates a series of problems:

- *Simplistic test cases* are created. Hard-coded test cases generally verify basic scenarios, but do not stress the logic hard enough to cover complex interactions. It takes substantial effort to create a large number of test cases that thoroughly test the logic with multiple interactions and permutations.
- A *new test-case language* must be mastered in order to write effective hard-coded test cases. The user must learn a test-case language that allows manipulation of signals, monitoring events, and clocking.
- *Maintenance* of test buckets can be time-consuming. Many hard-coded test cases are timing- and signal-name-dependent. If a cycle is added within the logic, or the name of an interface signal changes, many test cases may have to be updated.
- A *catch-22* problem occurs with the designer writing the test cases to test his own logic. Consider a case in which a designer believes his logic accounts for all valid permutations that can occur. If he has in fact missed a case, how can he be expected to write a test case that will uncover his own oversight?

- *TIMEDIAG and GENRAND as irritator behaviorals*

TIMEDIAG and GENRAND are intended to address the challenges associated with performing quality designer-level verification. They can be used to solve the problems associated with hard-coded test cases:

- *Complex scenarios* are created under GENRAND. It drives many different timing diagram permutations at different times throughout a test case. Window conditions, created by changing the timing on the initiation of sequences, are stressed. This helps solve the catch-22 problem as well, because the designer no longer has to dream up different scenarios—GENRAND does it instead.
- *No new language* has to be learned. The timing diagram format is familiar to designers.
- *There is little maintenance* with TIMEDIAG and GENRAND. If a signal name changes or a timing sequence is updated, all that is required is the use of TIMEDIAG to update the sequence. There are no buckets of test cases to change.

TIMEDIAG and GENRAND employ the irritator behavioral model at the designer level. This methodology takes advantage of the less complex environment, which makes the checking algorithms simpler. The random driver algorithms are built into GENRAND, freeing the designer from concerns about invoking sequences.

TIMEDIAG is a timing diagram editor. The file that is created by TIMEDIAG, when read by GENRAND, can be thought of as roughly equivalent to the interface protocol handler of the irritator behavioral model. This TIMEDIAG file contains one or more timing diagrams that describe the interfaces to the logic under test. GENRAND is the driver portion of the verification model. After reading the TIMEDIAG file, GENRAND uses

pseudorandom algorithms to decide which interface actions to initiate. When results are returned, GENRAND checks the correctness of the outputs and takes appropriate action by initiating further stimulus, responding to outstanding requests, or flagging errors or miscompares.

TIMEDIAG

TIMEDIAG is a tool that facilitates the creation of a file containing the information needed by GENRAND to drive the interfaces to the logic under test. Graphically, this information appears as timing diagrams that the user creates and edits. A TIMEDIAG file can contain any number of timing diagrams. Since the timing diagram format is straightforward and familiar, TIMEDIAG is also a good source of interface documentation.

Each timing diagram, in its simplest form, is a matrix of cycles and signal names. The cycle numbers C0, C1, $\cdots$, C*N* span the top of the matrix. The signal names and bit ranges constitute the rows of the matrix. Each of the values within the matrix is simply the value of that signal on that cycle. Each signal name is identified as an input to the logic or an output from the logic. Input signals have their cycle values entered into the logic by the test driver (GENRAND), while output signals have their cycle values checked for correctness by the test driver.

TIMEDIAG supplies all of the basics needed to edit each timing diagram matrix. A user can add, delete, copy, move, or change signal names. Cycles can also be added, deleted, copied, or moved. Finally, each cycle value in the matrix can be edited or copied from another value. Each diagram represents a generalized interface sequence. A given timing diagram is considered generalized because TIMEDIAG does not require exact cycle timings or constants for cycle values (variables, signals, and functions are allowed). Instead, TIMEDIAG includes features that allow the user to specify a sequence that may cover many permutations of an interface protocol.

- *TIMEDIAG features*

Three basic features allow a user to create general timing diagrams: function values, limitors, and looping/recurring cycles.

Function values

TIMEDIAG allows for many different data types and functions to represent cycle values. While constant values are the simplest form, signal names and arithmetic operators can also be used to represent the value of a signal on a given cycle. Furthermore, a set of built-in functions such as the "random value" function and the "choose from a list of values" function are supplied. If none of these built-in functions adequately describes a cycle value, a user-defined function written in C can be dynamically linked into TIMEDIAG/GENRAND.

Limitors

A cornerstone of the representation of an interface is the statement of the time at which it is legal to initiate the timing diagram sequence. TIMEDIAG defines this as a limitor. Each timing diagram has a limitor that may be independent or may be coupled to other timing diagrams. The limitor concept is especially important when running simulation, because GENRAND uses the limitors to decide which timing diagrams it can legally initiate on any given cycle.

TIMEDIAG uses four select buttons to help the user define a limitor. The first button, *none*, states that the timing diagram sequence may occur at any time and may be pipelined—there are no restrictions on when a "none" timing diagram may be initiated. The second select button, the *condition limitor*, uses a Boolean expression to evaluate whether or not the timing diagram may be initiated. If the statement evaluates to "true" on a given cycle, the timing diagram sequence may be initiated. The Boolean expression may be simple (example: AVAILABLE = '1'*B*) or complex [example: X = '1'*B* & $(Y > 2|Z < 3)$]. The third select button, the *delay limitor*, states that there must be a gap of some number of cycles between the initiation of two instances of timing diagrams that share the same delay limitor variable. This is useful in cases where, for example, a two-cycle gap is required between commands. The last select button, the *max limitor*, prevents the timing diagram from having more than some number of outstanding instances. For example, if a timing diagram were not allowed to be pipelined, the max limitor would be used with a value of 1. An example limitor update screen appears in Figure 3 (shown later).

It is legitimate for more than one of the last three select buttons (condition, delay, and max) to be used in a single limitor. As an example, consider an interface that can accept a command but may require up to 20 cycles to respond to that command. If the logic has four command buffers to hold outstanding commands but needs two cycles between each command in order to load the buffer, the proper limitor would be a "max of 4" and a "delay of 2." This same logic might have an "unavailable" line as an output, which, when set to '1'*B*, meant that no commands could be accepted. The condition limitor with "UNAVAILABLE = '0'*B*" would also be used in that case.

Timing diagram limitors can be used to couple timing diagrams or to prevent other timing diagrams from initiating. Multiple timing diagrams can be "coupled" by

- Using the same signal in both condition limitors.
- Using the same delay variable in both delay limitors.
- Using the same outstanding variable in both max limitors.

The limitor is further defined with a probability slide bar. This value provides GENRAND with the desired likelihood of initiating the diagram on a given cycle when the limiting conditions are true.

Recurring/looping cycles

Recurring or looping cycles help describe an interface where the timing between events is not fixed. In general, a recurring cycle says, "wait in this state until some event occurs" or "wait in this state for some number of cycles." Therefore, a recurring cycle is defined by its end condition.

TIMEDIAG allows for three possible end conditions: 1) wait for a condition to occur; 2) loop for a fixed number of cycles; or 3) loop for a (bounded) random number of cycles.

GENRAND

The general random driver program (GENRAND) provides the stimulus and verification for logic simulation. GENRAND uses the TIMEDIAG file as input, and then interfaces with the simulator to drive inputs and check outputs. GENRAND uses random algorithms to choose which interface sequences it drives. The interface sequences that are actually driven are based on the generalized timing diagrams that describe the legal interface protocols.

GENRAND does not determine the sequences before run time. Instead, GENRAND runs along with the simulator, using the timing diagram limitors and a random number generator to decide whether or not a timing diagram will be initiated on a given cycle. Once initiated, GENRAND emulates the timing diagram sequence in the following cycles until the timing diagram is completed. GENRAND continues to drive multiple timing diagrams until it either detects an error or runs to a predetermined quiesce cycle. The effect of this is that multiple window conditions are tested throughout a successful test case.

- *Timing diagram instances*

GENRAND uses the TIMEDIAG file to learn the interface protocols. The program then initiates interface sequences as specified in the timing diagrams (each interface sequence is an "instance" of that timing diagram), at random intervals as allowed by interface protocol. In fact, the same timing diagram will probably be initiated multiple times in a given simulation run. The simulation cycle in which an instance is initiated is not decided by the cycle labels (C0, C1, etc.) in the timing diagram. It is randomly chosen using the limitor conditions and the slide bar probability. However, the cycle labels in the timing diagrams can be mapped to the starting simulation cycle of a given instance. An instance of a timing diagram becomes "outstanding" or "bookmarked" when GENRAND randomly decides to initiate that timing diagram and the protocol allows for it. This instance is then bookmarked at timing diagram cycle C0. The bookmark remains open (and the instance thus remains outstanding) through the following simulation cycles until the last cycle of the timing diagram is executed.

Under GENRAND simulation, pipelining of a given timing diagram can occur if the protocol allows it. Therefore, multiple instances of the same timing diagram can be bookmarked at once, with each possibly in a different portion (C*x* cycle) of the timing diagram. The random occurrence of instances creates the desired complex environment. In order to track the instances, GENRAND creates a file that includes statistics about the occurrences of each timing diagram during the course of a run. A trace of the timing diagram instances can optionally be created as well. These tools are very helpful for problem debugging, which can be performed by recreating the same test case (by reusing the initial 32-bit seed).

TIMEDIAG/GENRAND usage for S/390 CMOS processor development

With the verification effort for the S/390 G4 CMOS processor and L2 chips focused on the designer macro level [2], TIMEDIAG and GENRAND were used extensively for the S/390 G4 processor development. TIMEDIAG/GENRAND was used on about 50% of the L2 control macros and on about 25% of the processor control macros. The L2 usage was greater because of the "requester" orientation of the cache control chip.

An example of the designer-level verification using TIMEDIAG and GENRAND is described next.

- *Operand fetch control verification with TIMEDIAG/GENRAND*

Traditionally, among the most difficult portions of logic to verify have been the instruction unit's operand fetch controls (OFC). The OFC interfaces mainly with the instruction unit's address adder and operand buffer controls, and with the buffer control element (BCE). Instructions are decoded and sent to the address adder, where operand addresses and request commands are sent to the OFC. The OFC allocates operand buffers, requests the data from the BCE, then monitors the BCE response and the return of data to the operand buffers. **Figure 1** shows the flow of control through the OFC.

For the S/390 G4 CMOS processor, the OFC logic was tested using TIMEDIAG/GENRAND. The effort took one

person about two months to complete. Three weeks of initial setup were followed by five weeks of testing and timing diagram enhancements. The strategy used to create timing diagrams for OFC verification was to write separate timing diagrams for each type of request. There were three main categories of timing diagrams, with an overall total of fourteen timing diagrams. The first category was for initial operand buffer requests from the address adder. Five separate timing diagrams were used for these initial operand requests: fetch, store, store–fetch, storage-to-storage request part one, and storage-to-storage request part two. The second main category of timing diagrams were the BCE interface diagrams, of which there were also five. Each of these diagrams represented one of the possible manners in which the BCE could respond to the OFC request: immediate, immediate lost, delayed by two cycles, delayed by two cycles and lost, and delayed long. The final category of timing diagrams represented checking and miscellaneous functions. The four timing diagrams in this category were "always," continuation fetch checking, incremental fetch request checking, and operand buffer release checking. **Figure 2** shows the TIMEDIAG main window, which contained the fourteen timing diagrams.

• *Address adder initial request timing diagrams*

The five initial request timing diagrams emulated the request protocol from the address adder to the OFC (transfer 1 in Figure 1). After an instruction was decoded, the address adder generated the appropriate request type (fetch, store, store–fetch, storage-to-storage request part one, and storage-to-storage request part two). The operand fetch controls are not privy to the actual instruction that was decoded; the OFC has to know only the type of instruction. Furthermore, the implementation of the OFC allows for any sequence of requests, with the sole exception that a storage-to-storage request part one must be followed by part two of the storage-to-storage request. Therefore, the timing diagrams were set up to allow for fetch, store, store–fetch, and storage-to-storage part one to be chosen at random. When a storage-to-storage part one occurred, these four commands were locked out until the storage-to-storage part two timing diagram was initiated. The only other restrictions on the initiation of the address adder request timing diagrams were that only one request could be initiated per cycle, and that the following three OFC signal settings must be true:

```
aa_ofc_available=1

aa_ofc_block_req=0

aa_ofc_hold=0
```

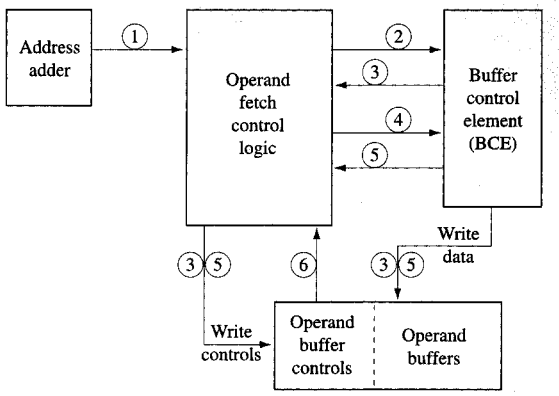

Figure 1

Flow of control through the OFC: (1) Address adder sends operand request to OFC after decoding; (2) OFC calculates operand address, buffer availability, and further buffer requirements, then sends first request to the BCE for processing; (3) BCE responds to operand request and sends data (for fetch-type operations only); (4) if further operand requests are required (long operand, page or line crossings), OFC sends follow-up requests to BCE; (5) BCE responds to follow-up requests (fetch data written to operand buffers); (6) operand buffer controls inform OFC logic when previously allocated buffer has been freed.

These signals indicated to the address adder the availability of the OFC to accept any further requests. This availability is based on the OFC's internal three-deep stack that holds unfinished operand requests. While the original source of all unfinished requests was the address adder, the stack could be filled by a single command if the address and length of the command were such that multiple BCE requests were required. Therefore, all of the address adder request timing diagrams had randomly selected addresses and lengths. This resulted in requests that spanned the logical possibilities, including line and page crossing as well as doubleword boundary crossing. Boundary crossings caused the OFC to calculate further BCE requests and operand buffer allocations.

The three signals that indicate OFC availability were the main limiting factor on the time at which GENRAND could initiate any of the initial request timing diagrams on a particular cycle. These signals are reflected in the conditional text of each of the initial request timing diagram limitors (see **Figure 3**). Four of the five initial request timing diagram limitors (excluding storage-to-storage part two) also had a gating signal condition [*td_need_ss2(0) = 0*] which prevented any of these four timing diagrams from initiating between a storage-to-storage part one and part two. This signal was a "program variable" in that it existed only in the timing diagrams and

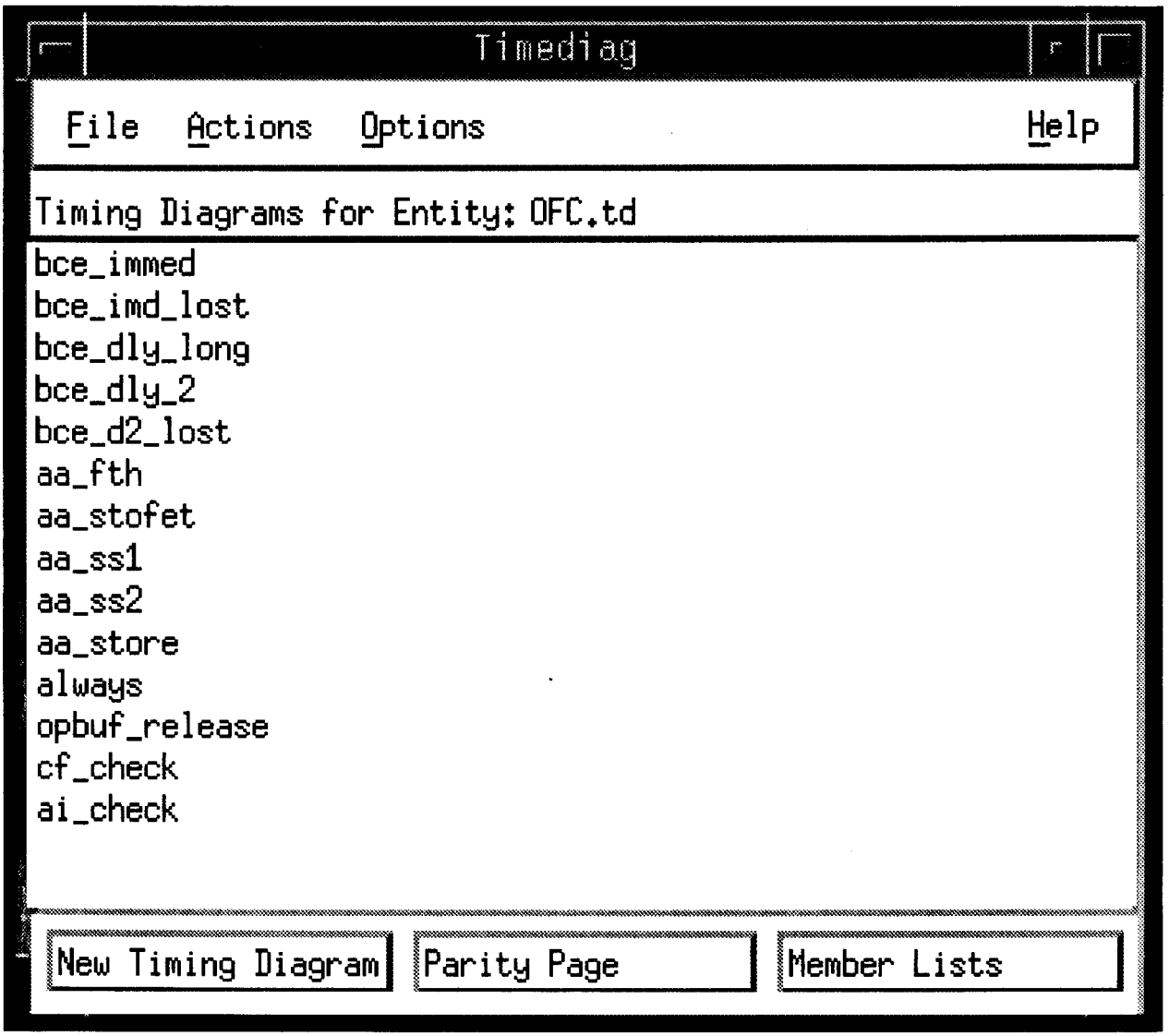

Figure 2

TIMEDIAG main window for OFC testing. The window lists all of the timing diagrams used for OFC verification. Any of the listed diagrams can be edited, copied, etc. The buttons at the bottom allow for creation of new timing diagrams, consolidation of parity input data, and creation of "lists of values" that can be used in the timing diagrams.

not in the real logic model. When GENRAND drove the simulation run, this signal was set to a '1'*B* in the first cycle of the storage-to-storage part one timing diagram and was reset in the first cycle of the part two timing diagram. This effectively locked out initiation of any other address adder request timing diagram during that period.

In addition to the signal conditions in the limitors of the address adder request timing diagrams, a delay limitor was used to prevent two different address adder requests from initiating on the same cycle. Had this case been allowed, two request timing diagrams would have logically ORed their requests together, thereby creating "garbage" on the request buses. Another "program variable" (*td_aa_lr*) was used in the delay field of all five request timing diagram limitors. When GENRAND chose to initiate one of the five request timing diagrams (the limitor conditions had to be true), *td_aa_lr* was immediately set to '1'*B* until the beginning of the next cycle, effectively locking out the other four timing diagrams for that cycle.

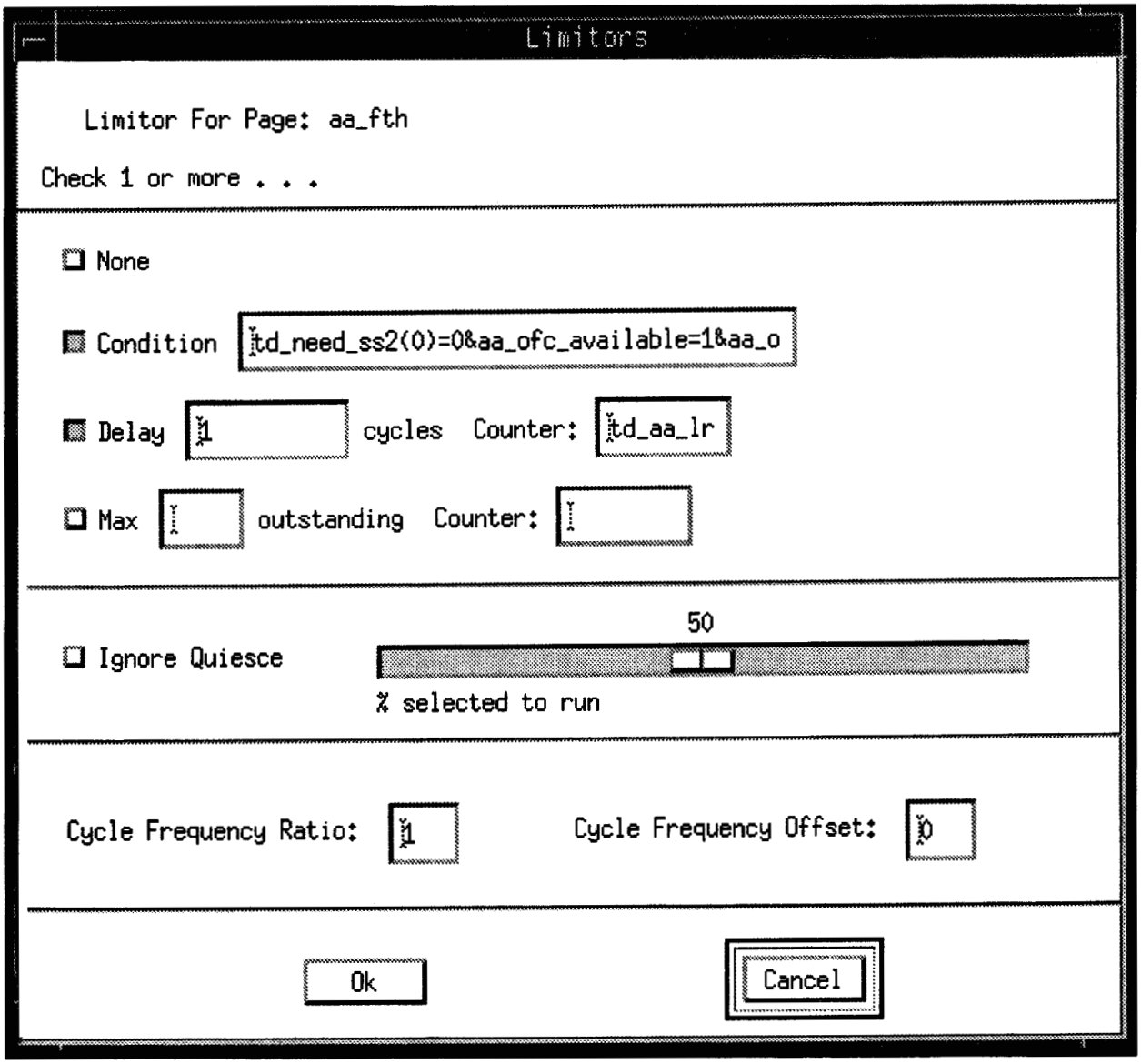

Figure 3

Limitor definition window for address adder request timing diagrams. The limitor definition window gives the user the ability to describe the conditions that dictate when the timing diagram can be initiated. The slide bar allows the user to change the probability of an instance of the timing diagram being run under GENRAND. This full conditional limitor reads: *td_need_ss2(0) = 0 & aa_ofc_available = 1 & aa_ofc_block_req = 0 & aa_ofc_hold = 0*, stating the conditions that must be true for an address adder request timing diagram to be initiated. For the storage-to-storage part two timing diagram, the *td_need_ss2(0)* condition was '1'. The delay portion of the limitor prevented a second address adder requester from being initiated in a given cycle.

The address adder request timing diagrams were fairly simple. They consisted of four cycles, one of which was a looping (recurring) cycle that waited for the BCE response and data to return (transfer 3 in Figure 1). The address adder fetch request timing diagram is shown in **Figure 4**. The first cycle of the timing diagram is the actual request cycle. Three "program variables" (*td_rnd_addr$*, *td_rnd_oplen$*, and *td_rnd_end_addr$*) are used to create and hold the random address and operand length for this request. GENRAND chose a valid random address and operand length for each instance.

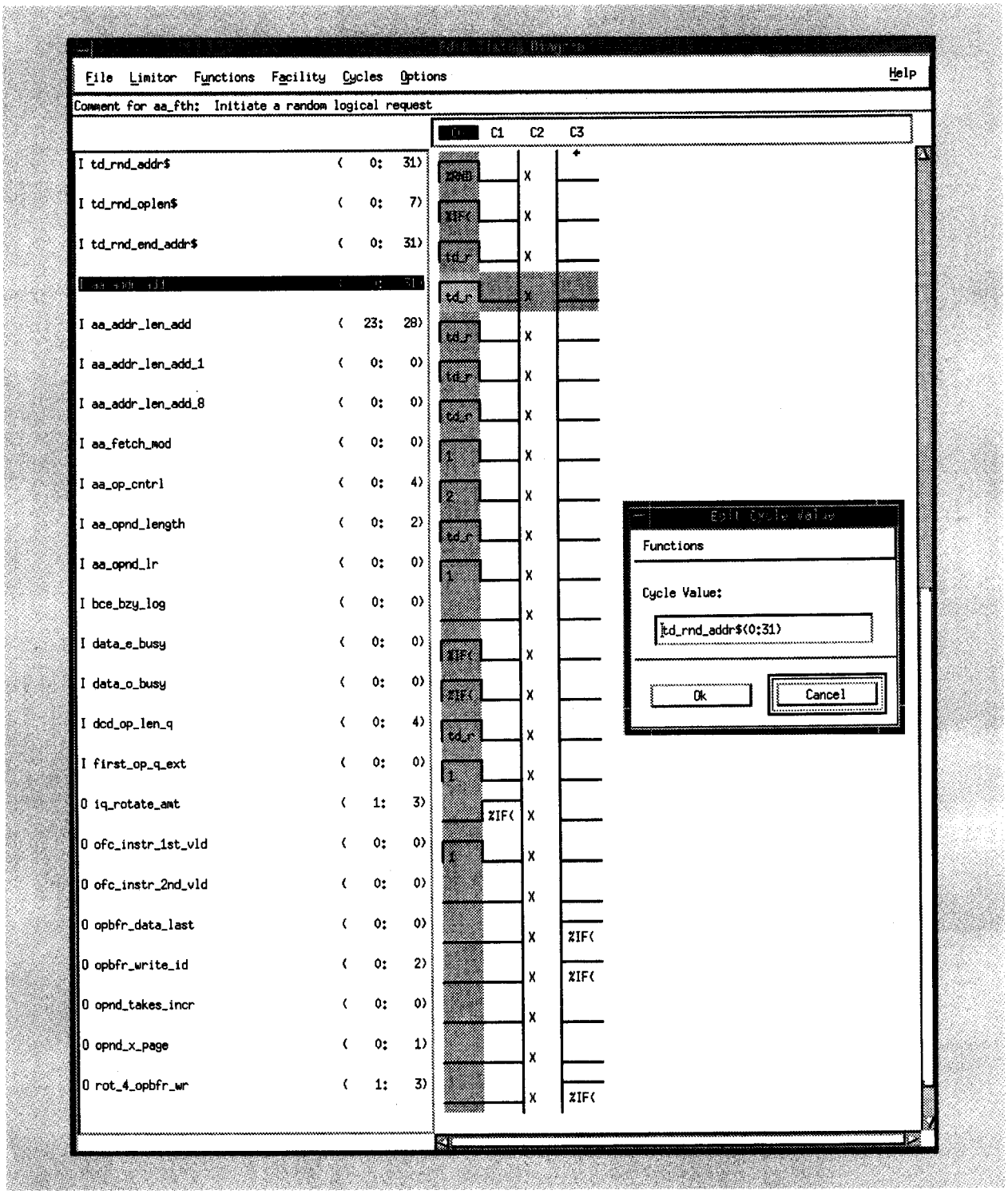

Figure 4

Address adder fetch timing diagram editing window. The timing diagram editing window, shown here with the cycle value update screen, allows the user to create timing diagrams in a familiar format. Only the first four characters of the cycle value are shown in the main timing diagram, but all characters can be seen in the cycle value update screen [currently selected is *aa_addr_add(0:31)* in cycle C0]. The vertical bars in the middle of the timing diagram represent a recurring/looping cycle. The 'X' cycle values represent a "don't care" condition.

The second cycle of the address adder request timing diagrams was used solely to check the value of *iq_rotate_amt*, which is calculated on the basis of the operand length and the low-order bits in the random address. The third cycle in the address adder request timing diagrams reflected the variable time needed to wait for the BCE to respond to the request. The final cycle performed much of the checking of the control lines from the OFC to the operand buffers when the data returned.

• *BCE interface timing diagrams*

After the OFC received an initial request from the address adder, one or more operand requests would be sent from the OFC to the BCE (transfers 2 and 4 in Figure 1). The number of required OFC-to-BCE operand requests depended on factors such as whether or not the operand crossed a doubleword boundary, line boundary, or page boundary, as well as the type of command (store or fetch operation). Fetch operations required additional OFC-to-BCE requests for every doubleword that the operand crossed. These additional requests were designated as "continuation fetches" because the BCE already has ownership of the line after the initial request is complete. If the fetch operand spans a line boundary, an "address increment" fetch is sent by the OFC to the BCE in order to bring the new line into the BCE and operand buffers. Page crossings for fetches required that an additional page alert command be sent to the BCE. For store operations, requests to the BCE were carried out only once for each line of the operand, so that the BCE gained exclusive access on the line(s) and prepared for store data.

The BCE can respond to the operand request in one of five ways. Each of these response possibilities was coded in an individual timing diagram. The five possible response types were as follows:

1. Immediate response: The BCE already has the data and, for fetches, will gate the data into the operand buffers two cycles after the request.
2. Immediate lost: The BCE is ignoring the request through the lack of acknowledgment two cycles after the request. The OFC must resend the request.
3. Delayed by two cycles: The BCE has the data, but cannot gate the data into the operand buffers until four cycles after the request.
4. Delayed by two cycles and lost: The BCE ignores the request through the lack of acknowledgment four cycles after the request. Two cycles after the request, the BCE has raised the *delayed_by_2* signal.
5. Delayed long: The BCE must access the data from the L2. The response will be delayed indefinitely. The BCE will raise the *req_advance* signal one cycle before the data are finally gated to the operand buffers.

To implement these responses, the five timing diagrams performed as a BCE behavioral. One of the five timing diagrams was randomly chosen on the cycle in which the OFC sent the request to the BCE. Therefore, the GENRAND program knew which type of response the OFC was going to get from the BCE as soon as the request was sent. This "insider knowledge" simplified the timing diagrams' ability to monitor the OFC requests for correctness. The early decision on which type of BCE response to send also trivialized the coding of the limitors for the five BCE response timing diagrams. Each diagram's limitor simply monitored the random choice (a value between 1 and 5, where 1 = Immediate, etc.).

The BCE response timing diagram, "delayed long," is shown in **Figure 5**. In this timing diagram, the BCE initially gives a *delayed_by_2* response to the OFC request, followed two cycles later by the *req_delayed* signal, indicating that the BCE must access the data line from the L2. The recurring cycle at C4 reflects the unpredictable wait time on the L2 access. For OFC verification purposes, this recurring cycle was coded to loop for between 4 and 15 cycles. (An actual value was chosen randomly by GENRAND each time the instance was encountered.) Finally, the *req_advance* and *log_req_done* (logical request done) were raised to indicate that the access was completed successfully. Two "program variables" (*td_log_req_done1* and *td_next_req_tossed*) were used to assist timing diagram verification control. The other signals in the timing diagram allowed for random variations in the sequence, which reflect all of the possible interface scenarios from the BCE.

• *Checking timing diagrams*

Four different checking timing diagrams were written to assist in the OFC verification. Each of these diagrams checked the outputs of the OFC for correctness under different conditions.

The "always" timing diagram was used to check for error conditions and invalid states on the OFC outputs. It also drove some miscellaneous inputs that were not related to the address adder or BCE interfaces. This timing diagram was designated "always" because the limitor was set to 100% probability and had no limiting conditions. This timing diagram contained only one cycle, so GENRAND initiated and completed the "always" timing diagram on every cycle. The "opbuf_release" timing diagram monitored the release of the six operand buffers. Since the release of the operand buffers was determined by logic outside the OFC and other modeled interfaces, this timing diagram also drove the operand buffer reset lines which indicated that the instruction corresponding to the buffer was complete. The number of cycles that passed before the buffer reset was varied. The final two timing diagrams verified the correctness of continuation fetches

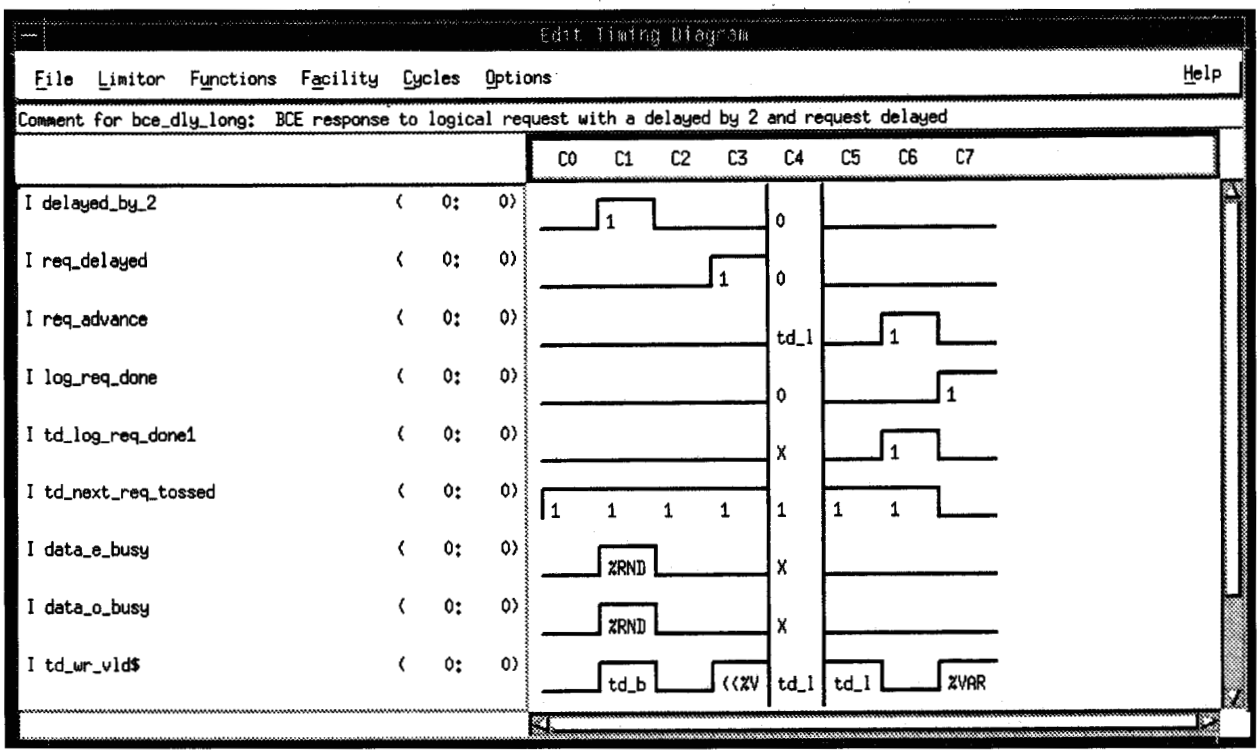

Figure 5

BCE timing diagram for "delayed long" response to OFC.

and address increment requests that were spawned by the OFC due to the length of the address adder request. Continuation fetches were checked in "cf_check," while address increment requests were verified in "ai_check." The main verification performed in both of these timing diagrams was for address and request correctness to ensure that the OFC was not sending erroneous requests to the BCE. As with the address adder request timing diagrams, "cf_check" and "ai_check" also verified the write controls from the OFC to the operand buffers.

• *GENRAND simulations of the OFC*

For early debugging, the initial GENRAND test cases verified each address adder request individually (GENRAND allows for disabling of timing diagrams by setting the limitor rate slide bar to 0%). Similarly, the five BCE response sequences were initially verified one at a time. This allowed GENRAND to create less "devious" test cases for initial debug of the OFC. Later, as each of the timing diagrams was verified, all combinations and sequences were enabled.

GENRAND simulation can create two different types of traces. The first is actually generated by the simulator and shows the real model signal and latch values on a cycle-by-cycle basis. This is the typical scoping tool provided with software simulators. The second trace is generated by the GENRAND program to track the timing diagram instances, as well as any "program variable" values. Together, the traces are valuable tools for debugging the design.

Errors, usually in the form of miscompares on OFC outputs, stopped the simulation run immediately. Miscompares were clearly displayed, along with information about which timing diagram and cycle had the miscompare with the actual versus expected signal values. Typical internal OFC problems were exposed as unexpected continuation fetches or incorrect addresses sent to the BCE.

• *Results*

Initially, short GENRAND test cases of less than 100 cycles uncovered six design errors. An additional ten errors were discovered over the next few weeks, using simulation runs of up to 10000 cycles. Furthermore, four performance problems were found using the TIMEDIAG/GENRAND methodology. Performance problems, which are transparent to architectural-level test cases, were found by TIMEDIAG/GENRAND testing

because the interfaces are closely monitored for expected timings on the outputs and responses. When the OFC was later incorporated into the unit-level and chip-level verification environments, no basic design problems were uncovered.

Using prior methods of designer-level verification, such as handwritten implementation tests, it would not have been possible to find many of the problems uncovered using TIMEDIAG/GENRAND. While static test cases verify general protocols or specific boundary conditions, the GENRAND algorithms test multiple cases of the protocols and allow for a full range of interaction among multiple interfaces.

The use of TIMEDIAG/GENRAND on the operand fetch controls was considered to be a time-to-market savings because the two months of time spent by one verification engineer early in the process returned a time savings in two areas. First, the knowledge of the OFC gained during the use of TIMEDIAG/GENRAND was carried forward to the processor-level verification, where the prior learning was used to assist in the integration of the OFC with the rest of the instruction unit and to debug areas that interfaced with the OFC. Second, processor-level and system-level verification were able to proceed more rapidly than previously because the OFC, as well as other designer-level tested logic designs, was functional soon after integration into the larger models.

Concluding remarks

It is expected that future improvements in TIMEDIAG/GENRAND will include the ability to cross-check the consistency of two interfacing functions' timing diagram files and allow for individual timing diagrams to be driven by different clocks. The expected improvements, along with the introduction of other low-level methodologies such as formal verification, should assist verification engineers in testing the ever-increasing complexity of processor and system designs.

Acknowledgments

The algorithms used in TIMEDIAG and GENRAND were refined through numerous discussions with Ed Kaminski and Dean Bair. Their help in coding TIMEDIAG and GENRAND brought the tool to life for the many engineers who have used it since. The assistance of Mark Check and John Liptay in creating the timing diagrams used on their operand fetch controls helped bring the complex logic into the mainstream simulation environments ahead of schedule.

*Trademark or registered trademark of International Business Machines Corporation.

References

1. D. F. Ackerman, M. H. Decker, J. J. Gosselin, K. M. Lasko, M. P. Mullen, R. E. Rosa, E. V. Valera, and B. Wile, "Simulation of the IBM Enterprise System/9000 Models 820 and 900," *IBM J. Res. Develop.* **36,** No. 4, 751–764 (July 1992).
2. B. Wile, M. P. Mullen, C. Hanson, D. G. Bair, K. M. Lasko, P. J. Duffy, E. J. Kaminski, Jr., T. E. Gilbert, S. M. Licker, R. G. Sheldon, W. D. Wollyung, W. J. Lewis, and R. J. Adkins, "Functional Verification of the CMOS S/390 Parallel Enterprise Server G4 System," *IBM J. Res. Develop.* **41,** No. 4/5, 549–566 (1997, this issue).

Received December 10, 1996; accepted for publication May 23, 1997

Bruce Wile *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (bwile@vnet.ibm.com).* Mr. Wile is currently a Senior Engineer and Verification Manager in S/390. He has worked in verification since joining IBM in 1985, and was the verification team leader for the S/390 G4 (CMOS 4) system. Mr. Wile's previous verification experiences included storage controller element simulation for the S/390 bipolar ES/9000 machines including the 6-way, 8-way, and 10-way multiprocessor systems. He was previously verification team leader for the 10-way IBM ES/9000 system. Mr. Wile received a B.S. in computer science from Pennsylvania State University in 1984. He received an IBM Excellence Award in 1992 and an IBM Team Award in 1993, and in 1995 an IBM Invention Achievement Award for inventions and patent submissions pertaining to the TIMEDIAG/GENRAND tool set.