

# Testing for Interconnect Crosstalk Defects Using On-Chip Embedded Processor Cores

Li Chen, Xiaoliang Bai, and Sujit Dey

Dept. ECE, University of California at San Diego, La Jolla, CA 92093-0407

{lichen, xibai, dey}@ece.ucsd.edu

## ABSTRACT

Crosstalk effects degrade the integrity of signals traveling on long interconnects and must be addressed during manufacturing testing. External testing for crosstalk is expensive due to the need for high-speed testers. Built-in self-test, while eliminating the need for a high-speed tester, may lead to excessive test overhead as well as overly aggressive testing. To address this problem, we propose a new software-based self-test methodology for system-on-chips (SoC) based on embedded processors. It enables an on-chip embedded processor core to test for crosstalk in system-level interconnects by executing a self-test program in the normal operational mode of the SoC. We have demonstrated the feasibility of this method by applying it to test the interconnects of a processor-memory system. The defect coverage was evaluated using a system-level crosstalk defect simulation method.

## 1. INTRODUCTION

The shrinking of feature sizes enables the integration of a large system comprising of multiple cores on a single chip. In core-based designs, a larger amount of core-to-core communications must be realized with long interconnects. As gate delay continues to decrease, the performance of interconnect is becoming increasingly important in achieving a high overall performance [1]. However, due to the increase of cross-coupling capacitance and mutual inductance, signals on neighboring wires may interfere with each other, causing excessive delay or loss of signal integrity. This effect, known as crosstalk, is more pronounced in deep-submicron technology [2][3]. While many techniques have been proposed to reduce crosstalk [4][5], due to the limited design margin and unpredictable process variations, the testing of crosstalk must be addressed in manufacturing testing.

Due to its timing nature, testing for crosstalk effect need to be conducted at the operational speed of the circuit-under-test. At-speed testing for GHz systems, however, is prohibitively expensive with external testers, as it requires testers with GHz performance. Moreover, with external testing, hardware access mechanisms are required for applying tests to interconnects deeply embedded in the system, which may lead to unacceptable area or performance overhead.

Compared with external testing, self-testing is a more feasible solution for at-speed crosstalk testing, as it does not

impose any performance requirement on the external tester. A built-in self-test (BIST) technique has been proposed in [6], in which an SoC tests its own interconnects for crosstalk defects using on-chip hardware pattern generators and error detectors. Although the amount of area overhead may be amortized for large systems, for small systems, the amount of relative area overhead may be unacceptable. Moreover, hardware-based self-test approach like the one proposed in [6] may cause over-testing, as not all test patterns generated in the test mode are valid in the normal operational mode of the system.

In this paper, we address the problem of testing system-level interconnects in embedded processor-based SoCs, which are the most dominant type of SoCs. In such SoCs, most of the system-level interconnects, such as the on-chip busses, are accessible to the embedded processor core(s). Based on this fact, we propose a software-based methodology that enables an embedded processor core in the SoC to test for crosstalk effects in these interconnects by executing a software program. Unlike previous embedded processor core-based self-testing methodologies [7][8][9][10], in which the embedded processor cores themselves are the subjects of testing, the focus of our methodology is on the testing of interconnects connecting the processor cores and other cores.

Compared with the hardware-based approaches for crosstalk testing, software-based self-test can be applied in the normal operational mode of the processor. Therefore, no extra hardware is needed. In addition, only the test patterns valid in the normal operational mode of the processor are applied. Thus, the system will not be over-tested.

In the rest of the paper, we first briefly describe the crosstalk fault model used during the development of the proposed method. The software-based self-test methodology for crosstalk defects is introduced in Section 3. In Section 4, we illustrate the proposed method in detail by constructing a self-test program for testing interconnects in a particular CPU-memory system. In Section 5, we validate the proposed method by extensive defect simulation using an HDL-level simulation framework consisting of a high-level crosstalk error model. Section 6 concludes the paper.

## 2. FAULT MODEL

During the development of the proposed self-test method, we used the Maximum Aggressor Fault (MAF) model [2] for modeling crosstalk effects. MAF model is a high-level representation of all physical defects and process variations that lead to crosstalk errors. It defines faults based on the resulting crosstalk error effects, including positive glitch ( $g_p$ ), negative glitch ( $g_n$ ), rising delay ( $d_r$ ), and falling delay ( $d_f$ ). The interconnect on which the error effect takes place is defined as the victim. All the other wires are designated aggressors, acting collectively to generate the glitch or delay error on the victim.

Figure 1 shows the signal transitions needed on the victim/aggressors to produce the strongest error effects on a victim wire. For example, to produce a positive glitch fault on Victim  $Y_i$ , a stable "0" is assigned to  $Y_i$ , whereas rising transitions are assigned to all aggressors. Each of these signal transitions, consisting of two consecutive test vectors, is defined as the Maximum Aggressor (MA) test for the corresponding MAF. For a set of  $N$ -interconnects, a total of  $4N$  faults need to be tested,

---

This work is supported in part by MARCO/DARPA Gigascale Silicon Research Center (GSRC) and the Semiconductor Research Corporation (SRC) through contract #98-TJ-648.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.  
Copyright 2001 ACM 1-58113-297-2/01/0006...\$5.00.

requiring  $4N$  unique MA tests. It has been proven in [2] that these MA tests are necessary and sufficient for covering all possible physical defects and process variations that can lead to any cross-coupling induced crosstalk error effect on any of the  $N$  interconnects.

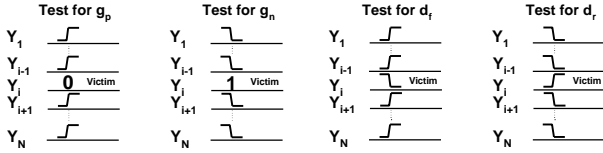


Figure 1. Maximum Aggressor Tests for Victim  $Y_i$

In the next section, we describe the general strategy for applying MA tests to system-level interconnects by executing a self-test program using an embedded processor.

### 3. TEST METHODOLOGY

In a core-based SoC consisting of embedded processor, memory and other cores, the address, data, and control busses are the main types of global interconnects, with which the embedded processors communicate with memory and other cores of the SoC via memory-mapped I/O (Figure 2). The data bus and the address bus are most susceptible to crosstalk defects, as they consist of a large number of long interconnects running in parallel. Thus, we focus in this paper on the testing of these busses.

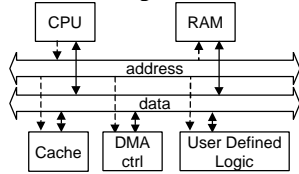


Figure 2. An SoC containing embedded processor cores

To test the address bus and the data bus with an embedded processor, our strategy is to let the processor execute a self-test program, with which the test vector pairs (as described in Section 2) can be applied to the appropriate bus in the normal functional mode of the system. In the presence of crosstalk-induced glitch or delay effects, the second vector in the vector pair becomes distorted at the receiver end of the bus. The processor can then store this error effect to the memory as a test response, which can later be unloaded by an external tester for off-chip analysis.

In this self-test approach, the loading of the self-test program and the unloading of the test response can be done with a low-speed tester. This prevents the corruption of the test program data or test response data due to crosstalk errors on the data or address busses, which can be activated at high speed. The self-test program itself is executed at-speed in the normal functional mode of the system without the monitoring of any external testers. Thus, with this approach, at-speed testing for crosstalk effects can be applied without a high-performance tester.

We next describe the general method for testing the data bus and the address bus.

#### 3.1 Testing Data Bus

For a bi-directional bus such as a data bus, crosstalk effects vary as the bus is driven from different directions. Thus, crosstalk tests need to be conducted in both directions [6]. Note that, however, to apply a test vector pair  $(v1, v2)$  in a particular bus direction, the direction of  $v1$  is irrelevant. Only  $v2$  needs to be applied in the specified direction. This is because the signal transition triggering the crosstalk effect takes place only when  $v2$  is being applied to the bus.

To apply a test vector pair  $(v1, v2)$  to the data bus from an SoC core to the CPU, the CPU first exchanges data  $v1$  with the core. The direction of the data exchange is irrelevant. For

example, if the core is the memory, the CPU may either read  $v1$  from the memory or write  $v1$  to the memory. The CPU then requests data  $v2$  from the core (a memory-read if the core is memory). Upon the arrival of  $v2$ , the CPU writes  $v2$  to memory for later analysis.

To apply a test vector pair  $(v1, v2)$  to the data bus from the CPU to an SoC core, the CPU first exchanges data  $v1$  with the core. The CPU then sends data  $v2$  to the core (a memory-write if the core is memory). If the core is memory,  $v2$  can be directly stored to an appropriate address for later analysis. Otherwise, the CPU must execute additional instructions to retrieve  $v2$  from the core and store it to memory.

#### 3.2 Testing Address Bus

To apply a test vector pair  $(v1, v2)$  to the address bus, which is a unidirectional bus from the CPU to an SoC core, the CPU first requests data from two addresses ( $v1$  and  $v2$ ) in consecutive cycles. In the case of a non-memory core, since the CPU addresses the cores via memory-mapped I/O,  $v2$  must be the address corresponding to the core. If  $v2$  is distorted by crosstalk, the CPU would be receiving data from a wrong address,  $v2'$ , which may be a physical memory address or an address corresponding to a different core. By keeping different data at  $v2$  and  $v2'$  (i.e.,  $mem[v2] \neq mem[v2']$ ), the CPU is able to observe the error and store it to memory for analysis. Figure 3 illustrates this process. For example, in the case where the CPU is communicating with a memory core, to apply test  $(0001, 1110)$  in the address bus from the CPU to the memory core, the CPU first reads data from address 0001. The CPU then reads data from address 1110. In the system with the faulty address bus, this address becomes 1111. If different data are stored at address 1110 and 1111 ( $mem[1110] = 0100$ ,  $mem[1111] = 1001$ ), the CPU would receive a faulty value from memory (1001 instead of 0100). This error response can later be stored to memory for analysis.

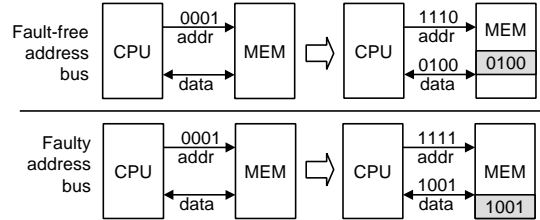


Figure 3. Testing the address bus

### 4. APPLICATION TO A CPU-MEMORY SYSTEM

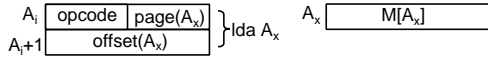
In this section, we illustrate the proposed method in detail by applying the guidelines introduced in Section 3 to a CPU-memory system. Since memory-mapped I/O is a common mechanism for CPU to communicate with other cores, the same methodology can be generalized for testing interconnects between the CPU and non-memory cores.

The particular CPU-memory system we used here consists of an 8-bit accumulator-based multi-cycle processor core with 23 instructions [11] and a 4K instruction/data memory. The CPU communicates with the memory via a 12-bit unidirectional address bus and an 8-bit bi-directional data bus.

#### 4.1 Testing Data Bus

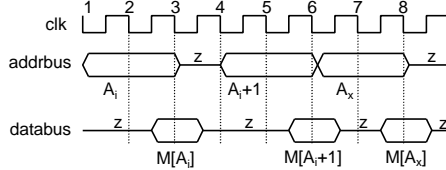
We chose to use the load instruction (LDA) of the processor to apply tests to the data bus. The load instruction takes a 12-bit address ( $A_x$ ) as operand and loads the content of this address ( $M[A_x]$ ) to the accumulator. As shown in Figure 4, the load instruction is stored as two bytes in the memory. The addresses of these two bytes are  $A_i$  and  $A_i+1$ , respectively. In the first byte, the

first 4 bits contain the opcode of the instruction. The last 4 bits contain the *page number* of  $A_x$ , which is the first 4 bits of  $A_x$ . The second byte in the load instruction contains the *offset* of  $A_x$ , which is the last 8 bits of  $A_x$ .



**Figure 4. Load instruction (LDA)**

Figure 5 shows the timing diagram of the load instruction. In the system under consideration, access to busses is controlled by tri-state buffers. When all tri-state buffers are disabled, the signal on the bus becomes high impedance (“z”). When “z” appears, we assume the bus holds the last defined value before “z”.



**Figure 5. Load instruction: timing diagram**

To fetch the instruction from the memory, the CPU first requests the first byte of the instruction by placing the address of the instruction,  $A_i$ , on the address bus. The memory responds by sending the content of  $A_i$  ( $M[A_i]$ ) to the CPU through the data bus. After decoding the first byte of the instruction, the CPU recognizes that the current instruction is a load instruction, which contains two bytes. Thus the CPU fetches the second byte of the instruction from memory ( $M[A_i+1]$ ). After receiving the complete instruction, which contains the data address  $A_x$ , the CPU fetches the content of  $A_x$  ( $M[A_x]$ ) from memory and places it into the accumulator.

During the execution of the load instruction, there are two signal transitions on the data bus, which can be used to apply the vector pair for crosstalk testing. The first transition is from  $M[A_i]$  to  $M[A_i+1]$ . As  $M[A_i]$  contains the opcode of the instruction, we are constrained when applying vector pairs using this transition. The second transition is from  $M[A_i+1]$  to  $M[A_x]$ , where  $M[A_i+1]$  contains the 8-bit address offset of the data to be loaded and  $M[A_x]$  contains the actual data to be loaded. We chose this transition over the first transition, as it does not come with any constraints.

To apply an arbitrary vector pair ( $v1$ ,  $v2$ ) to the data bus, we need to have  $M[A_i+1] = v1$  and  $M[A_x] = v2$ . Hence, we need to load from an address with a specific offset ( $v1$ ) containing a specific data ( $v2$ ). For example, to apply (00000000, 11110111), one of the tests for positive glitch faults, we may load from address 1110:00000000 (offset = 00000000), which contains data 11110111). If the test passes,  $v2$  (11110111) is loaded to the accumulator. If the test fails due to a positive glitch,  $v2$  becomes 11111111. Thus a wrong value is loaded into the accumulator. The error response can be collected by storing the accumulator content to a specific memory address, which can be checked by an external tester upon the completion of the tests. Thus, a two-instruction sequence is needed for applying this test: ( 1110:00000000, ), where the content of memory address 1110:00000000 must be set to 11110111, and is the memory address where the test response will be stored.

Similarly, the same strategy can be used to apply tests for other types of crosstalk faults (negative glitch, falling delay, and rising delay) on the data bus.

## 4.2 Testing Address Bus

During the execution of the load instruction, there are two transitions on the address bus. The first transition is the increment from  $A_i$  to  $A_i+1$ . The second transition is from  $A_i+1$  to  $A_x$ . Since

the first transition comes with a much more strict constraint than the second one, we choose to use the second transition to apply the vector pair.

To apply an arbitrary vector pair ( $v1$ ,  $v2$ ) to the address bus,  $A_i+1$  must be  $v1$  and  $A_x$  must be  $v2$ . Thus, the second byte of the instruction must be located at memory address  $v1$ , which implies that the instruction itself must be located at memory address  $v1-1$ . In addition, the data address accessed by the instruction must be  $v2$ .

For example, to apply (0000:00010000, 1111:11101111), one of the tests for falling delay faults, we place the load instruction at address 0000:00001111 ( $v1-1$ ), and load from address 1111:11101111 ( $v2$ ). If the test fails due to a falling delay defect,  $v2$  becomes 1111:11111111 and we would be loading from address 1111:11111111 instead of address 1111:11101111. To observe the error, we store different values at address 1111:11101111 and 1111:11111111 (e.g., 00000001 at 1111:11101111 and 00000000 at 1111:11111111). If the test passes, 00000001 is loaded to the accumulator. If the test fails, 00000000 is loaded instead. Again, the error response can be collected by storing the accumulator content to memory. Thus, a two-instruction sequence is needed for applying this test: ( 1111:11101111, ), where the instruction is placed at memory address 0000:00001111, and the contents of memory address 1111:11101111 and 1111:11111111 are set to 00000001 and 11111111, respectively.

Tests for rising delay faults can be applied in a similar manner. However, tests for positive glitch/negative glitch faults are considerably different. This is because all tests for positive glitch faults start with vector 0000:00000000 (Figure 1). To apply such a test using one instruction, the second byte of the instruction must be placed at memory address 0000:00000000. Moreover, to apply two vector pairs starting with the same vector (0000:00000000 in this case), two instructions need to be placed at the same memory location, causing an address conflict. This problem can be solved by utilizing the signal transition between two instructions, which we will not elaborate in this paper. Moreover, without losing any diagnostic information, we were able to compact test responses from different tests by instructions.

Depending on the actual instruction set of the processor core used in the SoC, the detailed constructs of the test programs may be different. Nonetheless, the general testing strategy we described here may be used to test the address/data busses between any CPU-memory pair. Moreover, since the cores in an SoC are often addressable by the CPU via memory-mapped I/O, the same test strategy can be applied to test address/data busses between any CPU-core pair.

## 5. VALIDATION

To validate the proposed software-based self-test methodology, we composed a complete test program for the CPU-memory system described in Section 4 and evaluated its defect coverage under an HDL-level defect simulation environment (Figure 6). During simulation, the CPU exercises the busses by executing the crosstalk test program stored in the memory. To simulate the effect of crosstalk on HDL-level, we used the high-level crosstalk error model proposed in [12]. Coded in HDL, the error model takes as input a parameter file containing the values of the coupling capacitance among interconnects. Given an input transition on the driver end of the bus, the error model determines whether a crosstalk error happens on the receiver end. Note that with this high-level crosstalk error model, we are able to take into account the effect of fault masking when evaluating defect coverage, since a crosstalk defect on the bus is indeed activated many times as the CPU executes the test program.

During the executing the test program, the CPU stores test response signatures to the memory. Upon the completion of the simulation, we determine whether the defect has been detected by

the test program by comparing the resulting response signature with its expected value. To estimate the defect coverage, the same defect simulation process is repeated on all defects from a pre-constructed defect library.

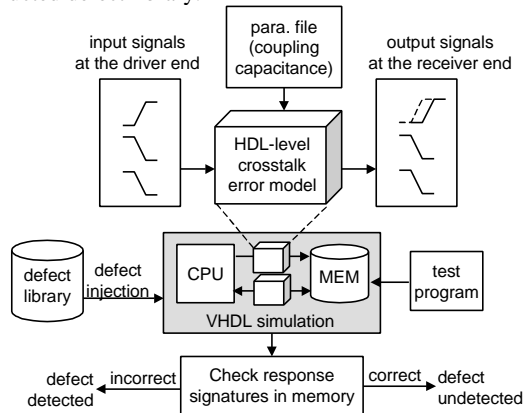


Figure 6. HDL-level defect simulation environment

To generate the defect library, we first randomly perturb the nominal values of coupling capacitances among interconnects according to a given defect distribution. Given the resulting perturbation, we use the criteria in [2] to determine whether the perturbation is large enough to be detectable by *any* tests. If so, we record the perturbation as a defect. The process is repeated until a satisfactory number of defects are generated. In this paper, we only consider crosstalk within the same bus when injecting defects. It is possible to inject defects causing crosstalk between two buses by treating them as one bus.

In our experiments, we used a Gaussian distribution to model the defect distribution in terms of the variation of capacitance values (in %). A 3 $\sigma$  point of 150% was chosen. A total number of 1000 defects were generated for each bus.

For the CPU-memory system described in Section 4, there are 64 MAFs on the 8-bit bi-directional data bus (8x4x2) and 48 MAFs on the 12-bit address bus (12x4). With the test program, we were able to apply 64 out of 64 MA tests for the databus and 41 out of 48 tests for the address bus. Some of the tests cannot be applied due to address conflicts – i.e., multiple tests compete for the same instruction address. This problem can be solved by separating conflicting tests into *multiple* test programs, which can be executed in different sessions. The total execution time of the programs is 1720 processor cycles. The size of the test program is proportional to the width of the busses, as a certain number of instructions are needed for testing for each MAF.

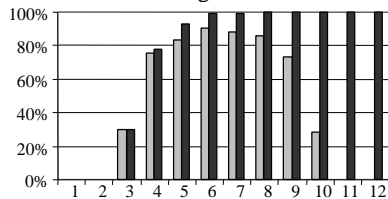


Figure 7. Crosstalk defect coverage of MA test programs

Figure 7 shows the individual and cumulative defect coverage obtained by applying each of the MA test for the address bus of the CPU-memory system. The horizontal axis indicates the MA test for each interconnect of the bus, with the  $i^{\text{th}}$  test being the MA test for the  $i^{\text{th}}$  interconnect. The individual defect coverages are shown in light gray, while the cumulative coverages are shown in dark gray. It can be seen that different MA tests have different levels of defect coverages, with the MA tests for the center interconnects having more coverage than the MA tests for the side interconnects. This is because the chance of a side interconnect

being defective is small, as the net coupling capacitance on a side interconnect is smaller than the one on a center interconnect (i.e., a much larger perturbation is needed to render the side interconnect defective). Figure 7 also shows that the MA tests combined together provide a 100% coverage of the crosstalk defects on the address bus interconnects.

Since the MA tests are necessary for detecting all detectable defects [2], in theory, some of the defects can only be detected by the missing tests. However, using our defect library, the defect coverage of the test program is 100% on both address and data busses. This is because a large overlap exists among the defects set detected by different MA tests. Of all the defects detectable by one MA test, only a tiny fraction cannot be detected by any other MA tests.

## 6. CONCLUSIONS

At-speed testing for crosstalk effects is expensive with external testers. Built-in self-test for crosstalk may result in high test overhead or over aggressive testing. To address these issues, we proposed a cost-effective method for testing system-level interconnects using embedded processor cores. By executing a self-test program, a processor is able to test the address and data busses through which it communicates with memory components. The same method can be applied for testing the interconnects between the processor and non-memory cores, as these cores are typically addressed by the processor via memory-mapped I/O. We have constructed an HDL level defect simulation environment to validate the proposed method and evaluate the defect coverage of any given test program. Experimental results show that a self-test program written following the proposed method is able to achieve its projected defect coverage.

## 7. REFERENCES

- [1] *The International Technology Roadmap for Semiconductors*, Semiconductor Industry Association, 1999.
- [2] M. Cuvillo, S. Dey, X. Bai, and Y. Zhao, "Fault modeling and simulation for crosstalk in system-on-chip interconnects," *1999 Int. Conf. Computer-Aided Design*, San Jose, CA, Nov. 1999, pp.297-303.
- [3] P. Nordholz, D. Treytnar, J. Otterstedt, H. Grabinski, D. Niggemeyer, and T.W. Williams, "Signal integrity problems in deep submicron arising from interconnects between cores," *Proc. 16<sup>th</sup> VLSI Test Symp.*, Monterey, CA, April 1998, pp.28-33.
- [4] H. Zhou and D.F. Wang, "Global routing with crosstalk constraints," *Proceedings of the 35<sup>th</sup> Design Automation Conference*, San Francisco, CA, USA, June 1998, pp.374-7.
- [5] A.B. Kahng, S. Muddu, E. Sarto, and R. Sharma, "Interconnect tuning strategies for high-performance ICs," *Proceedings Design, Automation and Test in Europe*, Paris, France, Feb. 1998, pp.471-8.
- [6] X. Bai, S. Dey, and J. Rajski, "Self-test methodology for at-speed test of crosstalk in chip interconnects," *Proc. 37<sup>th</sup> Design Automation Conf.*, Los Angeles, CA, June 2000, pp.619-24.
- [7] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. Computer-Aided Designs*, vol.20, no.3, March 2001.
- [8] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Proc. Int. Test Conf.*, Washington DC, Oct. 1998, pp. 990-999
- [9] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," *Proc. 17<sup>th</sup> VLSI Test Symp.*, Dana Point, CA, April 1999, pp. 34 – 40.
- [10] K. Radecka, J. Rajski, and J. Tyszer, "Arithmetic built-in self-test for DSP cores," *IEEE Trans. Computer-Aided Design*, vol.16, no.11, Nov. 1997, pp. 1358 – 69.
- [11] Z. Navabi, *VHDL: Analysis and modeling of digital systems*, New York, McGraw-Hill, 1993.
- [12] X. Bai and S. Dey, "High-level crosstalk defect simulation for system-on-chip interconnects," *Proc. 19<sup>th</sup> VLSI Test Symp.*, Los Angeles, CA, April 2001.