On the Detection of Synchronization Errors

Ian G. Harris

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
1 (949) 824-8842

harris@ics.uci.edu

CECS Technical Report 04-13
May, 2004

On the Detection of Synchronization Errors

Ian G. Harris

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
1 (949) 824-8842

harris@ics.uci.edu

CECS Technical Report 04-13
May, 2004

Abstract

Concurrent systems, either hardware or software, are notoriously difficult to design correctly in large part due to the complexities of nondeterministic execution. A concurrent system can perform many different correct computations for a given input sequence because the absolute order of execution is dependent on factors which cannot be known at design/compile time. Synchronization constructs are used to restrict the set of possible computations to only correct computations, but insertion of synchronization constructs is a manual and error-prone task. The detection of synchronization errors is made difficult because the manifestation of an error can depend on operation timing which can change between executions. We define a class of synchronization errors and define the timing requirements to ensure the detection of these errors. We provide a coverage metric which can be used to determine whether or not a given test execution is sufficient to detect the defined class of synchronization errors.

1 Introduction

Timing is an important part of the correctness of hardware/software systems. Although timing correctness and functional correctness are often evaluated separately, they are equally important aspects of a system's behavior. The significance of timing derives from the applications which use hardware/software systems. Many applications are embedded controllers which perform a time-sensitive activity such as an automatic braking system in a car in which a timing failure can be life threatening. Other applications include soft timing constraints such as a video display system, which must process video frames at some minimum rate to maintain the illusion of continuous motion for the user. Systems like this may not be life critical but timing failures result in a loss output quality and a loss of revenue when potential customers purchase competing products.

We will loosely define timing correctness as the ability of a system to produce a result within a predefined time limit. This can be contrasted with functional correctness which describes the ability of a system to produce a correct result, regardless of time. By this definition it is possible for a system to have correct timing but not correct function if an incorrect result is produced but it is produced on time. Strict timing goals have always been a part of the hardware design process but the general software community traditionally considers performance as a secondary goal. Evidence for the low degree of importance placed on timing requirements in software is the lack of explicit timing constructs in software languages. Because timing in hardware must be well controlled, all hardware description languages have features which allow designers to explicitly state timing relationships between events. Embedded software however shares the need for strict timing requirements in part because embedded software must interact directly with hardware. Hardware/software timing covalidation requires the use of techniques, which can be applied in both the hardware and software domains.

Hardware/software systems are typically built from a number of concurrently executing processes, which must coordinate to complete system tasks. The degree of concurrency directly impacts system timing and so it must be considered together with timing analysis. Concurrency is particularly relevant in the context of testing because systems containing concurrent execution are much more difficult to design than purely sequential systems. The difficulty stems largely from the fact that many different interleavings of concurrent operations are feasible for a given input sequence. This greatly increases the number of control flow possibilities making it more difficult for a human to follow. As a result, the aspects of system design which involve concurrency are the source of a disproportionately large number of design errors. The importance of concurrency in hardware/software codesign and the difficulty of concurrent design make the detection of concurrency-related design errors a serious problem.

The main reason for the difficulty in analyzing timing and concurrency is the presence of nondeterminism in the execution sequence. That is, there may exist many correct instruction execution sequences for a given system input sequence. Nondeterminism is a useful design tool because it allows the designer to ignore detailed instruction sequences which do not impact functional correctness. Several major sources of nondeterminism exist in hardware/software systems.

Operating System Scheduling - The scheduling algorithms applied by most operating systems are nondeterministic because they employ runtime information that cannot be known prior to execution. This impacts the instruction sequence directly and also the performance. Scheduling algorithms are typically designed to optimize the average case schedule, but the variation in timing between schedules may be significant. In practice, this type of nondeterminism is sometimes avoided by foregoing the use of an operating system entirely and implementing the changes of control flow directly into the application processes.

Microarchitectural Scheduling - Microprocessors often use dynamic scheduling techniques such as scoreboarding and Tomasulo's algorithm to improve performance through instruction reordering. In practice this type of nondeterminism may be avoided by using simple embedded processors that do not employ dynamic scheduling.

Memory Hierarchy - Using multiple levels of memory hierarchy makes memory delay nondeterministic because it depends on the hierarchy level at which the required data is found. Variable memory delays changes timing but it does not directly impact instruction sequencing, although it may be used in scheduling. For example, an out-of-order processor may decide to delay the execution of an instruction because the data it requires may be in main memory rather than cache. It is possible to eliminate this type of nondeterminism by using only one level of memory hierarchy but the performance will suffer. If the system can be designed to use no more memory than is contained in the L1 cache, then using only one level of hierarchy is feasible.

Nondeterminism is a problem for testing and validation because traditional (i.e. sequential) testing approaches assume that the correct execution sequence can be compared to a single known correct sequence to determine if the system executed correctly under test. In the presence of nondeterminism, a system with a design error has the potential to produce an incorrect execution sequence, but there is no guarantee whether or not that will occur. As a result, it is entirely possible that a system with a design error could produce correct sequences during testing and produce an incorrect sequence later after the system is deployed for use. Another test problem associated with the existence of multiple, correct instruction sequences is that all correct sequences must be specified for comparison during test. This increases memory required for test, which is an issue in hardware test, and it increases the time required to perform comparisons to check correctness.

Nondeterminism is managed in concurrent programs through the use of synchronization methods that restrict scheduling options to ensure correct operation. For example, if two processes cannot access some shared data at the same time, synchronization primitives must be added to the code to disallow the concurrent scheduling of operations which access the shared data. The task of using synchronization primitives in concurrent code is complicated and highly error prone. This chapter focuses on the errors involved in the synchronization between concurrent processes. We describe the most common methods of interprocess synchronization and then we describe the types of errors which commonly occur and their effects on behavior. We discuss the detection requirements of synchronization errors and present a fault model which can be used as a coverage metric to indicate the ability of a given test sequence to detect synchronization errors.

2 Synchronization Techniques

Any model for concurrent computation must enable process interaction of two forms [1]:
Contention - Two processes competing for the same resource.
Communication - Two processes passing information from one to the other.

Both types of process interaction depend on the ability to perform synchronization. Synchronization can be defined as the task of limiting the allowable interleavings between the execution of multiple processes. For example, managing contention typically requires that the execution of critical sections of code in two processes is mutually exclusive. Ensuring mutual exclusion necessitates synchronization because an interleaving of the processes should not be possible if it includes both processes executing their critical regions at the same time. Communication also requires synchronization because the existence of communication implies a data dependency between processes, which cannot be violated. For example, if process X sends data to process Y, then the part of process Y which uses the data cannot execute until after process X has computed and sent the data.

Synchronization is accomplished by forcing processes to agree that a certain event has taken place. The occurrence of the event is used as a *synchronization point* around which the allowable interleavings can be constrained. There are several synchronization techniques used in hardware and software languages that are summarized here.

Event synchronization identifies some changes in system state (such as a signal changing value) to be an event that synchronizes a process. Events may come from outside of the system or from other processes. Event synchronization provides two primitives, the *wait* primitive that causes a process to wait for an event, and the *post* primitive that causes the event to occur. The placement of the wait and post primitives in each process defines the synchronization points. The wait primitive can be synchronous, which causes the invoking process to be blocked until the event occurs, or the wait may be asynchronous which does not cause the process to block. In the asynchronous case some type of event handler must be provided to be executed when the event does eventually occur. Event synchronization with synchronized wait is the common technique in hardware description languages such as Verilog and VHDL.

The *semaphore* technique [2] introduces two synchronization primitives called P and V which operate on natural numbers called semaphores. The semaphores are visible to all communicating processes and events on semaphores are used for synchronization. V(s) increments the value of s while P(s) tests the value of s and decrements the value of s if it is greater than 0. If the value of s is equal to 0 then P(s) will block, suspending the execution of the process invoking P(s) until the value of s is greater than 0. A key property of the P and V primitives is that they are atomic, meaning that once they are initiated they cannot be interrupted until they are complete. If multiple processes invoke P or V at the same time then the executions occur sequentially in an arbitrary order. When a semaphore is incremented while there are several processes suspended by invoking P(s) on the semaphore, the processes are chosen to complete the P operation in an arbitrary order. Using semaphores, the incrementing and decrementing of semaphores are the events whose occurrence is agreed upon by all communicating processes. Possible interleavings are restricted by invoking P in a process to suspend it until V is invoked on the same semaphore by another process. The synchronization points are defined by the location of the invocations of P and V in the processes.

A *monitor* [3,4] is an object whose access is limited to only one process at a time. All data inside the object is private and can only be accessed using the access functions of the class. Only one access function can be executed at a time. If a process attempts the access the monitor while it is being accessed, the process is suspended until the current access is complete. To accomplish synchronization, a monitor defines a set of condition variables. A wait operation is defined to cause a process to suspend until an event occurs on a condition variable, and a signal operation causes an event to occur on a condition variable. The locations of the invocations of the wait and signal operations are the synchronization points. The use of condition variables is similar to synchronous event synchronization.

Unlike the event, semaphore, and monitor synchronization methods, *message-based communication* does not assume that processes share memory. Instead, data is transferred between processes using the send and receive operations. It is well known however that message-passing and shared memory communication schemes are equivalent. Any concurrent system implemented using one communication technique can also be implemented using the other. The send and receive operations can be either blocking or non-blocking, allowing the emulation of a range of synchronization methods. For example, the use of a non-blocking send and a blocking receive is equivalent to the synchronous event synchronization method used in most hardware description languages.

Another method of synchronization in a message passing architecture is the use of *remote procedure calls* (RPC) [5], or the more general *rendezvous* [6]. RPC enables a client process to invoke a function in another server process by using a procedure call which is similar to a normal procedure call within a single process. The client uses a send function to pass the name of the remote procedure to be invoked and the parameters of the procedure. The server process must invoke an accept function to indicate that it is ready to execute the requested procedure and the caller's send function must block until the request is accepted. Once the server process has completed the procedure it uses a return function to return the results to the caller. The caller must invoke a receive function to receive the function results from the server and the server's return function will block until the results are received. RPCs are asymmetric because the client can call procedures in the server but the server cannot make requests of the client. Rendezvous is a generalized version of RPC which allows processes to invoke procedures from each other in a symmetrical way.

3 A Class of Synchronization Errors

Each synchronization method requires the programmer to manually insert synchronization points into each process and this insertion process is a common source of errors in concurrent system design. We define this class of design errors and we describe the detection of errors in this class.

To understand synchronization errors its necessary to establish the relationship between the placement of synchronization points in a system description and the behavior of that system. We will use the simple concurrent system depicted in Figure 1 to describe the impact of synchronization point placement on behavior. Figure 1 shows the outline of a system with two concurrent processes, a producer process and a consumer process. The code for the producer and consumer processes shown is minimal in order to highlight only the features of interest to this discussion. The producer sends data to the consumer through a variable *sh_data* which both processes share access to. The statement in the producer which assigns a value to the sh_data variable is referred to as a *definition* of sh_data, def(sh_data). The statement in the consumer which reads the value of sh_data is a *use* of sh_data, use(sh_data). In this example event synchronization is assumed and the variable *ready* is used to indicate when the consumer is ready to receive new data. The synchronization point shown in the producer is the *wait (ready)* statement and the synchronization point in the consumer is the *ready <= 1* statement.

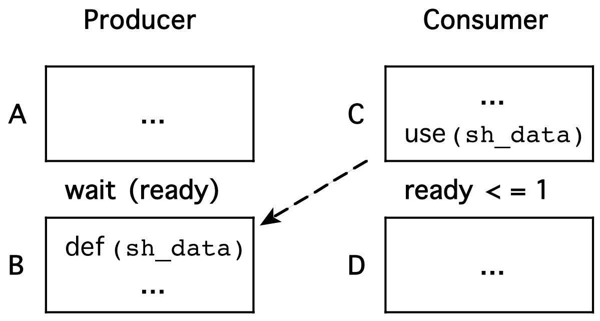

Fig. 1.Synchronization in a Producer/Consumer Example

The producer and consumer descriptions in Figure 1 are partitioned into four sequential blocks A, B, C, and D as determined by the placement of the synchronization points in the description. Each sequential block is a sequence of contiguous sequential instructions containing no synchronization points. The placement of the synchronization points establishes a dependency between sequential block B and C, so block B cannot execute until block C has completed. The dependency between B and C is needed to enforce the data dependency between the definition and use of the sh_data variable in blocks B and C. The definition and use of the sh_data variable represents a potential write-after-read (WAR) hazard, which is prevented using synchronization points.

If the synchronization points are incorrectly placed, the sequential blocks are redefined and the WAR hazard may occur. If the wait statement in the producer is placed after the definition of sh_data, it is possible for the definition to occur before the use, causing the incorrect value of sh_data to be used. The same problem occurs if the *ready* <= *1* statement were accidentally placed before the use of the sh_data variable.

Incorrect placement of synchronization points can allow data dependency errors to exist between processes, but the manifestation of these errors depends on the scheduling. If the wait statement is misplaced, the definition of sh_data in the producer could be incorrectly scheduled before the use in the consumer and the error could be detected. However, it is possible that the wait could be misplaced but that the definition is never scheduled before the use during testing, and so the error is not detected. Such an undetected error could still manifest itself later in the product lifetime. This example demonstrates that the detection of synchronization errors depends on the schedule, which is in general nondeterministic.

4 A Fault Model for Synchronization Errors

To facilitate testing for synchronization errors a model is needed which enumerates all of the synchronization errors that will occur in an arbitrary design. The large number of potential design errors makes direct enumeration infeasible, so the model must be an abstract one. We refer to this model as a *fault model* which defines a set of faults for an arbitrary design. Each fault described by the model represents a set of potential errors in a design and the detection of all faults ensures the detection of all errors or the type covered by the fault model.

According to their manifestation in time, design faults can be grouped into two classes, static faults whose observation is independent of absolute event timing, and timing faults whose observation depends on a specific timing of events on shared data. The observation of a static fault depends on the sequence of test pattern application, but not the absolute time of the application of each pattern. An example of a static fault is the replacement of the expression x = y +1 with the incorrect expression x = y + 2, where the variable x is shared between two processes. Once this fault is activated, its effects can be observed at any time before the signal x is redefined. A timing fault exists when a signal is assigned to the correct value, but the event occurs at the incorrect time. A timing fault will cause a signal value to endure for the incorrect length of time. The timing fault effect can be observed only during the incorrect time period which we will refer to as the *error span*. The difference between static faults and timing faults is that a timing fault is active during only a subset of the time period between two definitions, while a static fault is active during the entire time period between two definitions. Such a timing fault is a result of a synchronization error because correct synchronization would prevent a process from using the value of x until it has the correct value. Correct synchronization can be seen as a mechanism to make concurrent execution independent of timing. In the case of incorrect synchronization, the behavior of the system becomes timing dependent and timing faults can occur.

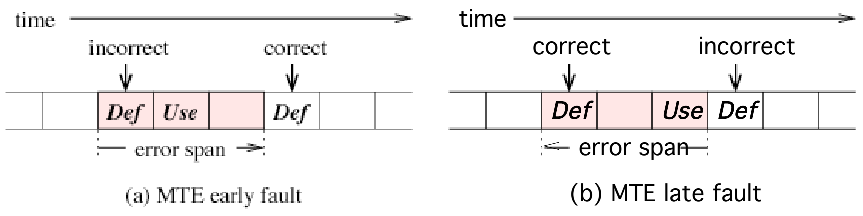

Fig. 2. Two types of MTE faults

We refer to these faults as Mis-Timed Event (MTE) faults because they are caused by timing relationships between access to shared data. The MTE fault model [7] is derived from the all-definition-use pair metric [8] developed for the testing of sequential programs. An MTE fault is associated with each definition-use pair for each shared variable or object. The existence of an MTE fault indicates that the associated definition and use occur in the wrong sequence due to incorrect synchronization between the processes accessing the shared data. Two types of MTE faults can exist between a definition-use pair: MTE_{early} indicates that the definition occurs earlier than the correct time, and MTE_{late} indicates that the definition occurs later than the correct time. Figure 2a shows an MTE_{early} fault whose error span extends forward in time from the incorrect time step, and Figure 2b shows an MTE_{late} fault whose error span extends backward in time from the incorrect time step. Figure 2 assumes a discrete time model which is common in hardware simulation but the concept applies to continuous time as well.

4.1 Detection of Synchronization Faults

An MTE fault associated with a signal is detected only if there is a use of the shared variable inside the error span of the fault, as shown in Figure 2. The error span extends from the erroneous time step to the correct time step. Unfortunately, the precise position of the error span is not known since simulation of the faulty circuit reveals only the erroneous time step. It is clear, however, that the error span must extend, either forward or backward in time, from the erroneous time step. In order to ensure that a use occurrence is within the error span of a fault, the use occurrence must be close to the corresponding definition occurrence in time. Also, a use occurrence must exist both earlier than the definition and later than the definition to detect both late and early MTE faults. These circumstances exist in Figures 2a and 2b where, in each case, the use occurrence is immediately adjacent to the erroneous time step. The detection of the MTE_{late} fault is accomplished by the use-definition (ud) pair where the use occurs before the erroneous definition time step, and the MTE_{early} fault is detected by the definition-use (du) pair where the use occurs after the erroneous definition time step.

To ensure the detection of an MTE fault the associated du or ud pair must be close in time. An *error span threshold* value d must be provided to define the maximum time difference between the definition and the use which is assumed to detect the fault. The error span threshold determines the sensitivity of the testing process to small perturbations in timing, so a small threshold ensures high sensitivity. However, if the threshold is too small then the timing behavior of the system may make the fault undetectable. For example, the operating system may impose a minimum delay to perform a context switch between two processes and if the error span threshold is smaller than that minimum delay then MTE faults between the two processes will be undetectable. Identification of the minimum allowable threshold requires a solution to the minimum time separation problem [9] which is known to be NP-complete. We assume that the error span threshold is provided by a design/test engineer who has knowledge of the system timing behavior.

4.2 Fault Coverage Computation

The practical use of a fault model requires that there be an efficient procedure to determine which faults are detected when a given test sequence is applied to a design. We implemented MTE coverage computation using Verilog PLI which allows MTE coverage to be computed for designs described in Verilog. The computation algorithm contains three main steps.

1. *du/ud pairs identification*- in this step, we generate lists of du/ud pairs for each signal by analyzing the behavioral description. Since a signal can be defined and used in multiple modules in a hierarchical design, it may have different names in different modules. In order to catch any occurrence of the signal, we first find the top module in which the signal is first declared as a signal. Along the connection down to the submodules we recursively find all modules using the signal. In each such a module, we locate the definition and use statements and register a callback function for each occurrence statement. After locating all definition and use occurrence, all du pairs sharing the same use are associated with that use occurrence and all ud pairs sharing the same definition are associated with that definition occurrence. The initial value of time separation for each du/ud pair is set to a negative number, which is updated during the simulation.

2. *Simulation* - The behavioral description is simulated with the test sequence. For each signal, there is a record of the current definition which defines the current value of the signal. The record includes the location of the current definition and the time step when the definition occurred. The record of the current definition is updated every time a new definition occurs. During the simulation, a callback function is called when a statement with definition or use occurrence is executed. The callback function then records the time step of the occurrence. For a definition occurrence, the callback function updates the record of the current definition and calculates the new value of time separation for each ud pair associated with the current definition. If the new value of separation is less than the old one, the callback function updates the time separation of the ud pair. For a use occurrence, the callback function calculates the value of time separation for the du pair from the current definition to the use and updates the value of separation if the new value is smaller.

3. *MTE fault coverage analysis* - In this step, the simulation results are analyzed and the MTE fault coverage is calculated for a range of thresholds. For a given threshold, the MTE fault coverage is the ratio of the number of the du/ud pairs executed within threshold to the number of all du/ud pairs. Since the value of the threshold strongly affects the fault coverage, the coverage result is presented by a curve rather than a number. The resulting curve shows the trend of coverage over a range of threshold values.

5 Experimental Results

We experimented with the MTE fault analysis tool using four industry designs that were provided to us with functional test sequences which we evaluated for MTE coverage. Each benchmark was provided in Verilog and the Cadence Verilog-XL simulator was use to gather coverage information. The first benchmark is an implementation of type 1 ATM Adaptation Layer protocol that abstracts ATM layer from higher level communication protocols. The type 1 AAL protocol is used to provide Constant Bit Rate (CBR) service such as conventional voice service and existing leased line service. The second benchmark is an implementation of a four ports data switch that contains an arbiter and four ports to receive and send data. Each port sends requests for the internal bus and the arbiter chooses one to allow access. The third benchmark is an implementation of Dual Tone Multi-frequency (DTMF) receiver. DTMF signals are the control tones generated by standard touch tone telephones. Pressing a key causes the telephone to generate a pair of tones, one from the high frequency group, and one from the low frequency group. To detect the tones, DTMF receiver utilizes Goertzel's algorithm to calculate the frequency response at the DTMF center frequencies. Once calculated, the frequency response is analyzed determine which DTMF digit was found. The fourth benchmark is a simple RISC CPU core.

Bench.	**Lines**	**Blocks**	**Signals**	**Pairs**	**Stmt**	**MTE**
AAL1	2068	54	22	1732	0.70	0.10
Switch	1269	28	29	200	0.93	0.65
Risc8	2302	50	37	1032	0.60	0.54
DTMF	8383	77	17	262	0.54	0.46

Table 1. Benchmark Information and Coverage Summary

Table 1 summarizes the benchmark information and the coverage results. The first 4 columns in order show the benchmark names, lines of Verilog code, number of *always* (concurrent) blocks, and number of signals used. The number of signals is relevant because each signal acts as a shared variable and the MTE faults are associated with definitions and uses of the signals. The fifth column contains the number of du/ud pairs for all of the signals which is also the number of MTE faults in the design. The sixth column shows the statement coverage produced during test application, the fraction of statements covered during simulation. The seventh column shows the maximum MTE coverage values, the fraction of MTE faults which are detected by the test sequence. The MTE coverages reported in Table 1 are maximal because the error span threshold was set to the maximum value.

Fig. 3. AAL1 MTE fault coverage distribution

Detailed MTE coverage results for the AAL1 benchmark are shown in Figures 3 and 4. Figure 2 shows the variation in MTE coverage over a range of error threshold values. As the error threshold increases the MTE coverage rises because larger separation between du/ud pairs is allowed. The maximum MTE coverage in this example is quite low, only 0.10, and MTE coverage is used to identify weaknesses in the test sequence and locate coverage holes. Upon examining the benchmark, we find that 87% of the du/ud pairs are associated with one signal rec_seq that has 28 definitions and 27 uses. This results in 1512 du/ud pairs. However, only 35 out of these 1512 pairs were executed during simulation. The rec_seq signal is defined 28 times in the *receiver.rec_CPU.chk1* module whose statement coverage is only 0.22, and is used 23 times in the *receiver.rec_cpu.fsm1* module whose statement coverage is 0.40. Therefore, most of the definitions and uses are not executed and the MTE fault coverage of this signal is only 0.02 which results in the overall low MTE coverage on the design. Figure 3 shows the distribution of MTE coverage without consideration of signal rec_seq and the coverage increases to 0.63. The test sequence needs to be enhanced to cover the du/ud pairs involving uses in the *receiver.rec_cpu.fsm* module. In the case the MTE fault model identifies a weakness in the test sequence and provides direction on how the sequence should be changed to improve the completeness of testing.

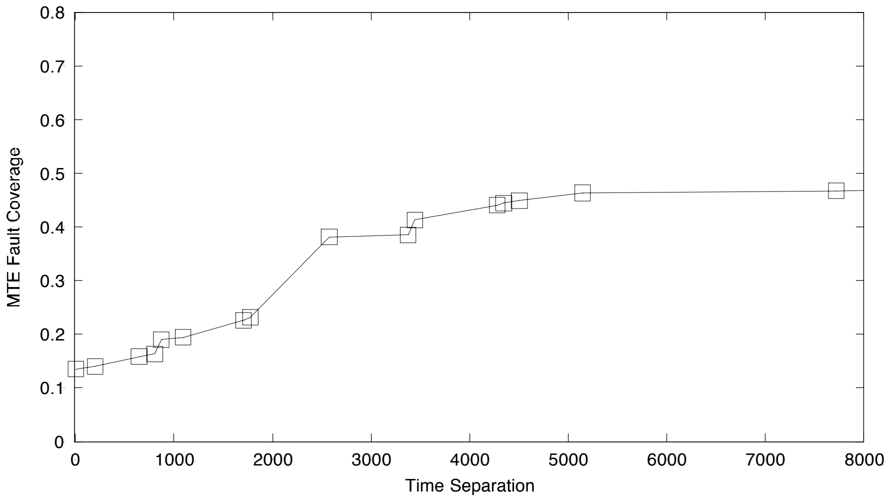

Fig. 4. AAL MTE coverage without the rec_seq signal

6 Conclusions

As hardware/software codesign is increasing applied for the design of embedded applications, both functional and timing correctness of these designs becomes more important. Process synchronization is manually intensive and has proven to be a difficult task due to the complex interprocess dependencies that must be considered. Many synchronization methods are in use, which attempt to ease the synchronization task by providing abstract primitives for use by the designer. However, it is not possible to enable efficient design while completely hiding the intricacies of the synchronization task from the designer. As a result, synchronization is likely to be a difficult and error prone process for the foreseeable future. A significant body of research is dedicated to hardware and software validation [10], but little of this existing research focuses on the synchronization problem. Given the inherent difficulty in synchronization, further research in this area can be expected.

1.7 Acknowledgements

Work on this report was supported in part by the National Science Foundation under grant number 0204134.

References

1. Ben-Ari M (1990) Principles of Concurrent and Distributed Programming. Prentice Hall International (UK) Ltd
2. Dijkstra EW (1968) Cooperating Sequential Processes, Programming Languages 43-112
3. Hoare CAR (1974) Monitors: An Operating System Structuring Concept. Communications of the ACM 17(10):549-557
4. Brinch Hansen P (1973) Operating System Principles. Prentice Hall. Englewood Cliffs, NJ
5. Brinch Hansen P (1978) Distributed Processes: A Concurrent Programming Concept. Communications of the ACM 21, 11 November, 934-941
6. Hoare CAR (1978) Communicating Sequential Processes. Communications of the ACM 21, 8 August 666-667
7. Zhang Q, Harris IG (2001) A Validation Fault Model for Timing-Induced Functional Errors. International Test Conference 813-820
8. Rapps S, Weyuker EJ (1985) Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering SE-11, 4:367-375

9. Chakraborty S, Dill DL (1997) Approximate Algorithms for Time Separation of Events. International Conference on Computer-Aided Design 190-198
10. Harris IG (2003) Fault Models and Test Generation for Hardware-Software Covalidation. IEEE Design and Test of Computers 20(4):40-47