Module 8

Testing of Embedded System

Lesson
38
Testing Embedded Systems

Instructional Objectives

After going through this lesson the student would be able to

- Distinguish between the terms testing and verification
- Describe the common types of faults that occur in embedded systems
- Explain the various types of models that are used to represent the faults
- Describe the methodology of testing systems with embedded cores
- Distinguish among terms like DFT, BIST and on-line testing
- Explain the need and mechanism of Automatic Test Pattern Generation in the context of testing embedded hard-ware software systems

Testing Embedded Systems

1. Introduction

What is testing?

- Testing is an organized process to verify the behavior, performance, and reliability of a device or system against designed specifications.
- It ensures a device or system to be as defect-free as possible.
- Expected behavior, performance, and reliability must be both formally described and measurable.

Verification vs. Testing [1]

- Verification or debugging is the process of removing defects ("bugs") in the design phase to ensure that the synthesized design, when manufactured will behave as expected.
- Testing is a manufacturing step to ensure that the manufactured device is defect free.
- Testing is one of the *detective* measures, and verification one of the *corrective* measures of quality.

Verification	**Testing**
Verifies the correctness of design.	Verifies correctness of manufactured system.
Performed by simulation, hardware emulation, or formal methods.	Two-part process: 1. Test generation: software process executed once during design. 2. Test application: electrical tests applied to hardware.

Performed once prior to manufacturing.	Test application performed on every manufactured device.
Responsible for quality of design.	Responsible for quality of devices.

What is an "embedded system"?

Embedded systems are electronically controlled system where hardware and software are combined [2-3]. These are computers incorporated in consumer products or other devices to perform application-specific functions. The enduser is usually not even aware of their existence. Embedded systems can contain a variety of computing devices, such as microcontrollers, application-specific integrated circuits, and digital signal processors. Most systems used in real life as power plant system, medical instrument system, home appliances, air traffic control station, routers and firewalls, telecommunication exchanges, robotics and industrial automation, smart cards, personal digital assistant (PDA) and cellular phone are example of embedded system.

Real-Time System

Most, if not all, embedded systems are "real-time". The terms "real-time" and "embedded" are often used interchangeably. A real-time system is one in which the correctness of a computation not only depends on its logical correctness, but also on the time at which the result is produced.

- In hard real time systems if the timing constraints of the system are not met, system crash could be the consequence. For example, in mission-critical application where failure is not an option, time deadlines must be followed.
- In case of soft real time systems no catastrophe will occur if deadline fails and the time limits are negotiable.

In spite of the progress of hardware/software codesign, hardware and software in embedded system are usually considered separately in the design process. There is a strong interaction between hardware and software in their failure mechanisms and diagnosis, as in other aspects of system performance. System failures often involve defects in both hardware and software. Software does not “break” in the traditional sense, however it can perform inappropriately due to faults in the underlying hardware, as well as specification or design flaws in either the hardware or the software. At the same time, the software can be exploited to test for and respond to the presence of faults in the underlying hardware. It is necessary to understand the importance of the testing of embedded system, as its functions have been complicated. However the studies related to embedded system test are not adequate.

2. Embedded Systems Testing

Test methodologies and test goals differ in the hardware and software domains. Embedded software development uses specialized compilers and development software that offer means for debugging. Developers build application software on more powerful computers and eventually test the application in the target processing environment.

In contrast, hardware testing is concerned mainly with functional verification and self-test after chip is manufactured. Hardware developers use tools to simulate the correct behavior of circuit models. Vendors design chips for self-test which mainly ensures proper operation of circuit models after their implementation. Test engineers who are not the original hardware developers test the integrated system.

This conventional, divided approach to software and hardware development does not address the embedded system as a whole during the system design process. It instead focuses on these two critical issues of testing separately. New problems arise when developers integrate the components from these different domains.

In theory, unsatisfactory performance of the system under test should lead to a redesign. In practice, a redesign is rarely feasible because of the cost and delay involved in another complete design iteration. A common engineering practice is to compensate for problems within the integrated system prototype by using software patches. These changes can unintentionally affect the behavior of other parts in the computing system.

At a higher abstraction level, executable specification languages provide an excellent means to assess embedded-systems designs. Developers can then test system-level prototypes with either formal verification techniques or simulation. A current shortcoming of many approaches is, however, that the transition from testing at the system level to testing at the implementation level is largely ad hoc. To date, system testing at the implementation level has received attention in the research community only as coverification, which simulates both hardware and software components conjointly. Coverification runs simulations of specifications on powerful computer systems. Commercially available coverification tools link hardware simulators and software debuggers in the implementation phase of the design process.

Since embedded systems are frequently employed in mobile products, they are exposed to vibration and other environmental stresses that can cause them to fail. Some embedded systems, such as those in automotive applications, are exposed to extremely harsh environments. These applications are preparing embedded systems to meet new and more stringent requirements of safety and reliability is a significant challenge for designers. Critical applications and applications with high availability requirements are the main candidates for on-line testing.

3. Faults in Embedded Systems

Incorrectness in hardware systems may be described in different terms as defect, error and faults. These three terms are quite bit confusing. We will define these terms as follows [1]:

Defect: A defect in a hardware system is the unintended difference between the implemented hardware and its intended design. This may be a process defects, material defects, age defects or package effects.

Error: A wrong output signal produced by a defective system is called an error. An error is an “effect” whose cause is some “defect”. Errors induce failures, that is, a deviation from appropriate system behavior. If the failure can lead to an accident, it is a *hazard*.

Fault: A representation of a “defect” at the abstraction level is called a fault. *Faults* are physical or logical defects in the design or implementation of a device.

3.1 Hardware Fault Model (Gate Level Fault Models)

As the complexity and integration of hardware are increasing with technology, defects are too numerous and very difficult to analyze. A fault model helps us to identify the targets for testing and analysis of failure. Further, the effectiveness of the model in terms of its relation to actual failures should be established by experiments. Faults in a digital system can be classified into three groups: design, fabrication, and operational faults. Design faults are made by human designers or CAD software (simulators, translators, or layout generators), and occur during the design process. These faults are not directly related to the testing process. Fabrication defects are due to an imperfect manufacturing process. Defects on hardware itself, bad connections, bridges, improper semiconductor doping and irregular power supply are the examples of physical faults. Physical faults are also called as defect-oriented faults. Operational or logical faults are occurred due to environmental disturbances during normal operation of embedded system. Such disturbances include electromagnetic interference, operator mistakes, and extremes of temperature and vibration. Some design defects and manufacturing faults escape detection and combine with wearout and environmental disturbances to cause problems in the field.

Hardware faults are classified as stuck-at faults, bridging faults, open faults, power disturbance faults, spurious current faults, memory faults, transistor faults etc. The most commonly used fault model is that of the "stuck-at fault model" [1]. This is modeled by having a line segment stuck at logic 0 or 1 (stuck-at 1 or stuck-at 0).

Stuck-at Fault: This is due to the flaws on hardware, and they represent faults of the signal lines. A signal line is the input or output of a logic gate. Each connecting line can have two types of faults: stuck-at-0 (s-a-0) or stuck-at-1 (s-a-1). In general several stuck-at faults can be simultaneously present in the circuit. A circuit with n lines can have $3^n - 1$ possible stuck line combinations as each line can be one of the three states: s-a-0, s-a-1 or fault free. Even a moderate value of n will give large number of multiple stuck-at faults. It is a common practice, therefore to model only single stuck-at faults. An n-line circuit can have at most *2n* single stuck-at faults. This number can be further reduced by fault collapsing technique.

Single stuck-at faults is characterized by the following properties:

1. Fault will occur only in one line.
2. The faulty line is permanently set to either 0 or 1.
3. The fault can be at an input or output of a gate.
4. Every fan-out branch is to be considered as a separate line.

Figure 38.1 gives an example of a single stuck-at fault. A stuck-at-1 fault as marked at the output of OR gate implies that the faulty signal remains 1 irrespective of the input state of the OR gate.

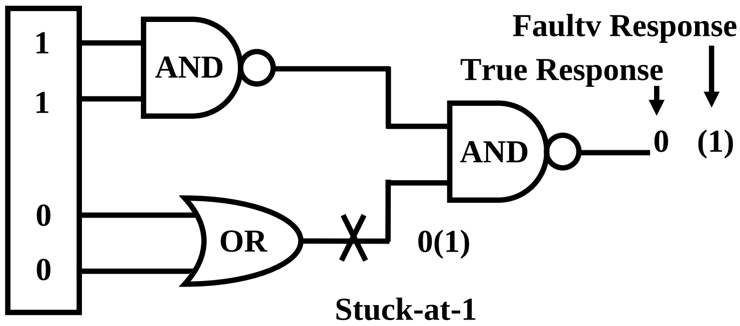

Fig. 38.1 An example of a stuck-at fault

Bridging faults: These are due to a short between a group of signal. The logic value of the shorted net may be modeled as 1-dominant (OR bridge), 0-dominant (AND bridge), or intermediate, depending upon the technology in which the circuit is implemented.

Stuck-Open and Stuck-Short faults: MOS transistor is considered as an ideal switch and two types of faults are modeled. In stuck-open fault a single transistor is permanently stuck in the open state and in stuck-short fault a single transistor is permanently shorted irrespective of its gate voltage. These are caused by bad connection of signal line.

Power disturbance faults: These are caused by inconsistent power supplies and affect the whole system.

Spurious current faults: that exposed to heavy ion affect whole system.
Operational faults are usually classified according to their duration:

Permanent faults exist indefinitely if no corrective action is taken. These are mainly manufacturing faults and are not frequently occur due to change in system operation or environmental disturbances.
Intermittent faults appear, disappear, and reappear frequently. They are difficult to predict, but their effects are highly correlated. Most of these faults are due to marginal design or manufacturing steps. These faults occur under a typical environmental disturbance.

Transient faults appear for an instant and disappear quickly. These are not correlated with each other. These are occurred due random environmental disturbances. Power disturbance faults and spurious current faults are transient faults.

3.2 Software-Hardware Covalidation Fault Model

A *design error* is a difference between the designer's intent and an executable specification of the design. Executable specifications are often expressed using high-level hardware-software languages. Design errors may range from simple syntax errors confined to a single line of a design description, to a fundamental misunderstanding of the design specification which may impact a large segment of the description. A *design fault* describes the behavior of a set of design errors, allowing a large set of design errors to be modeled by a small set of design faults. The majority of covalidation fault models are behavioral-level fault models. Existing covalidation fault models can be classified by the style of behavioral description upon which the models are based. Many different internal behavioral formats are possible [8]. The covalidation fault models currently applied to hardware-software designs have their origins in either the hardware [9] or the software [10] domains.

3.2.1 Textual Fault Models

A textual fault model is one, which is applied directly to the original textual behavioral description. The simplest textual fault model is the statement coverage metric introduced in software testing [10] which associates a potential fault with each line of code, and requires that each statement in the description be executed during testing. This coverage metric is accepted as having limited accuracy in part because fault effect observation is ignored. Mutation analysis is a textual fault model which was originally developed in the field of software test, and has also been applied to hardware validation. A *mutant* is a version of a behavioral description which differs from the original by a single potential design error. A *mutation operator* is a function which is applied to the original program to generate a mutant.

3.2.2 Control-Dataflow Fault Models

A number of fault models are based on the traversal of paths through the contol data flow graph (CDFG) representing the system behavior. In order to apply these fault models to a hardware-software design, both hardware and software components must be converted into a CDFG description. Applying these fault models to the CDFG representing a single process is a well understood task. Existing CDFG fault models are restricted to the testing of single processes. The earliest control-dataflow fault models include the branch coverage and path coverage [10] models used in software testing.

The branch coverage metric associates potential faults with each direction of each conditional in the CDFG. The branch coverage metric has been used for behavioral validation for coverage evaluation and test generation [11, 12]. The path coverage metric is a more demanding metric than the branch coverage metric because path coverage reflects the number of control-flow paths taken. The assumption is that an error is associated with some path through the control flow graph and all control paths must be executed to guarantee fault detection.

Many CDFG fault models consider the requirements for fault activation without explicitly considering fault effect observability. Researchers have developed observability-based behavioral fault models [13, 14] to alleviate this weakness.

3.2.3 State Machine Fault Models

Finite state machines (FSMs) are the classic method of describing the behavior of a sequential system and fault models have been defined to be applied to state machines. The commonly used fault models are *state coverage* which requires that all states be reached, and *transition coverage* which requires that all transitions be traversed. State machine *transition tours*, paths covering each transition of the machine, are applied to microprocessor validation [15]. The most significant problem with the use of state machine fault models is the complexity resulting from the state space size of typical systems. Several efforts have been made to alleviate this problem by identifying a subset of the state machine which is critical for validation [16].

3.2.4 Application-Specific Fault Models

A fault model which is designed to be generally applicable to arbitrary design types may not be as effective as a fault model which targets the behavioral features of a specific application. To justify the cost of developing and evaluating an application-specific fault model, the market for the application must be very large and the fault modes of the application must be well understood. For this reason, application-specific fault models are seen in microprocessor test and validation [17,18].

3.3 Interface Faults

To manage the high complexity of hardware-software design and covalidation, efforts have been made to separate the behavior of each component from the communication architecture [19]. Interface covalidation becomes more significant with the onset of core-based design methodologies which utilize pre-designed, pre-verified cores. Since each core component is pre-verified, the system covalidation problem focuses on the interface between the components. A case study of the interface-based covalidation of an image compression system has been presented [20].

4. Testing of Embedded Core-Based System-on-Chips (SOCs)

The system-on-chip test is a single composite test comprised of the individual core tests of each core, the UDL tests, and interconnect tests. Each individual core or UDL test may involve surrounding components. Certain operational constraints (e.g., safe mode, low power mode, bypass mode) are often required which necessitates access and isolation modes.

In a core-based system-on-chip [5], the system integrator designs the User Defined Logic (UDL) and assembles the pre-designed cores provided by the core vendor. A core is typically hardware description of standard IC e.g., DSP, RISC processor, or DRAM core. Embedded cores represent intellectual property (IP) and in order to protect IP, core vendors do not release the detailed structural information to the system integrator. Instead a set of test pattern is provided by the core vendor that guarantees a specific fault coverage. Though the cores are tested as part of overall system performance by the system integrator, the system integrator deals the core as a black box. These test patterns must be applied to the cores in a given order, using a specific clock strategy.

The core internal test developed by a core provider need to be adequately described, ported and ready for plug and play, i.e., for interoperability, with the system chip test. For an internal test to accompany its corresponding core and be interoperable, it needs to be described in an commonly accepted, i.e., standard, format. Such a standard format is currently being developed by IEEE PI 500 and referred to as standardization of a core test description language [22].

In SOCs cores are often embedded in several layers of user-defined or other core-based logic, and direct physical access to its peripheries is not available from chip I/Os. Hence, an electronic access mechanism is needed. This access mechanism requires additional logic, such as a wrapper around the core and wiring, such as a test access mechanism to connect core peripheries to the test sources and sinks. The wrapper performs switching between normal mode and the test mode(s) and the wiring is meant to connect the wrapper which surrounds the core to the test source and sink. The wrapper can also be utilized for core isolation. Typically, a core needs to be isolated from its surroundings in certain test modes. Core isolation is often required on the input side, the output side, or both.

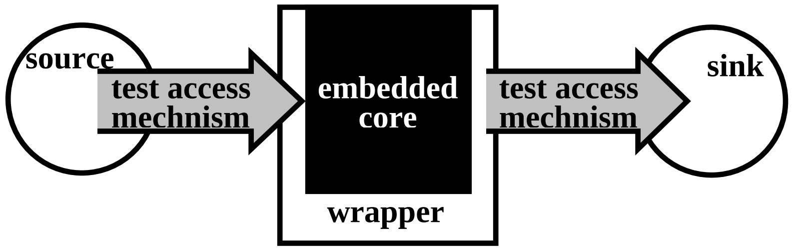

Fig. 38. 2 Overview of the three elements in an embedded-core test approach: (1) test pattern source, (2) test access mechanism, and (3) core test wrapper [5].

A conceptual architecture for testing embedded-core-based SOCs is shown in Figure 38.2 It consists of three structural elements:

1. Test Pattern Source and Sink

The test pattern source generates the test stimuli for the embedded core, and the test pattern sink compares the response(s) to the expected response(s). Test pattern source as well as sink can be implemented either off-chip by external Automatic Test Equipment (ATE), on-chip by Built-In Self-Test (or Embedded ATE), or as a combination of both. Source and sink do not need to be of the same type, e.g., the source of an embedded core can be implemented off-chip, while the sink of the same core is implemented on-chip. The choice for a certain type of source or sink is determined by (1) The type of circuitry in the core, (2) The type of pre-defined tests that come with the core and (3) Quality and Cost considerations. The type of circuitry of a certain core and the type of predefined tests that come with the core determine which implementation options are left open for test pattern source and sink. The actual choice for a particular source or sink is in general determined by quality and cost considerations. On-chip sources and sinks provide better accuracy and performance related defect coverage, but at the same time increase the silicon area and hence might reduce manufacturing yield.

2. Test Access Mechanism

The test access mechanism takes care of on-chip test pattern transport. It can be used (1) to transport test stimuli from the test pattern source to the core-under-test, and (2) to transport test responses from the core-under-test to the test pattern sink. The test access mechanism is by definition, implemented on-chip. Although for one core often the same type of' test access mechanism is used for both stimulus as well as response transportation, this is not required and various combinations may co-exist. Designing a test access mechanism involves making a trade-off between the transport capacity *(bandwidth)* of the mechanism and the test application cost it induces. The bandwidth is limited by the bandwidth of source and sink and the amount of silicon area one wants to spend on the test access mechanism itself.

3. Core Test Wrapper

The core test wrapper forms the interface between the embedded core and its system chip environment. It connects the core terminals both to the rest of the IC, as well as to the test access mechanism. By definition, the core test wrapper is implemented on-chip.
The core test wrapper should have the following mandatory modes.

- *Normal operation* (i.e., non-test) mode of' the core. In this mode, the core is connected to its system-IC environment and the wrapper is transparent.
- *Core test* mode. In this mode the test access mechanism is connected to the core, such that test stimuli can be applied at the core's inputs and responses can be observed at the core's outputs.
- *Interconnect test* mode. In this mode the test access mechanism is connected to the interconnect wiring and logic, such that test stimuli can be applied at the core's outputs and responses can be observed at the core's inputs.

Apart from these mandatory modes, a core test wrapper might have several optional modes, e.g., a *detach* mode to disconnect the core from its system chip environment and the test access mechanism, or a *bypass* mode for the test access mechanisms. Depending on the implementation of the test access mechanism, some of the above modes may coincide. For example, if the test access mechanism uses existing functionality, normal operation and core test mode may coincide.

Pre-designed cores have their own internal clock distribution system. Different cores have different clock propagation delays, which might result in clock skew for inter-core communication. The system-IC designer should take care of this clock skew issue in the functional communication between cores. However, clock skew might also corrupt the data transfer over the test access mechanism, especially if this mechanism is shared by multiple cores. The core test wrapper is the best place to have provisions for clock skew prevention in the test access paths between the cores.

In addition to the test integration and interdependence issues, the system chip composite test requires adequate test scheduling. Effective test scheduling for SOCs is challenging because it must address several conflicting goals: (1) total SOC testing time minimization, (2) power dissipation, (3) precedence constraints among tests and (4) area overhead constraints [2]. Also, test scheduling is necessary to run intra-core and inter-core tests in certain order not to impact the initialization and final contents of individual cores.

5. On-Line Testing

On-line testing addresses the detection of operational faults, and is found in computers that support critical or high-availability applications [23]. The goal of on-line testing is to detect fault effects, that is, errors, and take appropriate corrective action. On-line testing can be performed by external or internal monitoring, using either hardware or software; internal monitoring is referred to as *self-testing*. Monitoring is internal if it takes place on the same substrate as the circuit under test (CUT); nowadays, this usually means inside a single IC—a system-on-a-chip (SOC).
There are four primary parameters to consider in the design of an on-line testing scheme:

• *Error coverage* (*EC*): This is defined as the fraction of all modeled errors that are detected, usually expressed in percent. Critical and highly available systems require very good error detection or *error coverage* to minimize the impact of errors that lead to system failure.

• *Error latency* (*EL*): This is the difference between the first time the error is activated and the first time it is detected. *EL* is affected by the time taken to perform a test and by how often tests are executed. A related parameter is *fault latency* (*FL*), defined as the difference between the onset of the fault and its detection. Clearly, $FL \geq EL$, so when *EL* is difficult to determine, *FL* is often used instead.

• *Space redundancy* (*SR*): This is the extra hardware or firmware needed to perform on-line testing.

• *Time redundancy* (*TR*): This is the extra time needed to perform on-line testing.

An ideal on-line testing scheme would have 100% error coverage, error latency of 1 clock cycle, no space redundancy, and no time redundancy. It would require no redesign of the CUT, and impose no functional or structural restrictions on the CUT. To cover all of the fault types described earlier, two different modes of on-line testing are employed: *concurrent testing* which takes place during normal system operation, and *non-concurrent testing* which takes place while normal operation is temporarily suspended. These operating modes must often be overlapped to provide a comprehensive on-line testing strategy at acceptable cost.

5.1 Non-concurrent testing

This form of testing is either event-triggered (sporadic) or time-triggered (periodic), and is characterized by low space and time redundancy. Event-triggered testing is initiated by key events or state changes in the life of a system, such as start-up or shutdown, and its goal is to detect permanent faults. It is usually advisable to detect and repair permanent faults as soon as possible. Event-triggered tests resemble manufacturing tests.

Time-triggered testing is activated at predetermined times in the operation of the system. It is often done periodically to detect permanent faults using the same types of tests applied by event triggered testing. This approach is especially useful in systems that run for extended periods, where no significant events occur that can trigger testing. Periodic testing is also essential for detecting intermittent faults. Periodic testing can identify latent design or manufacturing flaws that only appear under the right environmental conditions.

5.2 Concurrent testing

Non-concurrent testing [23] cannot detect transient or intermittent faults whose effects disappear quickly. Concurrent testing, on the other hand, continuously checks for errors due to such faults. However, concurrent testing is not by itself particularly useful for diagnosing the source of errors, so it is often combined with diagnostic software. It may also be combined with non-concurrent testing to detect or diagnose complex faults of all types.

A common method of providing hardware support for concurrent testing, especially for detecting control errors, is a watchdog timer. This is a counter that must be reset by the system on a repetitive basis to indicate that the system is functioning properly. A watchdog timer is based on the assumption that the system is fault-free—or at least alive—if it is able to perform the simple task of resetting the timer at appropriate intervals, which implies that control flow is correctly traversing timer reset points.

For critical or highly available systems, it is essential to have a comprehensive approach to on-line testing that covers all expected permanent, intermittent, and transient faults. In recent years, built-in-self-test (BIST) has emerged as an important method for testing manufacturing faults, and it is increasingly promoted for on-line testing as well.

6. Test Pattern Generation

6.1 Test Plan

Test plans are generated to verify the device specification, which comprise of the decision on test type, fault coverage, test time etc. For example, the test pattern generator and response analyzer may reside on an automatic test equipment (ATE) or on-chip, depending on the test environment. In the case of production testing in an industry, ATE may be the option, while on-site testing may require on-chip testers (BIST).

6.2 Test Programming

The test program comprises modules for the generation of the test vectors and the corresponding expected responses from a circuit with normal behavior. CAD tools are used to automate the generation of optimized test vectors for the purpose [1,24]. Figure. 38.3 illustrates the basic steps in the development of a test program.

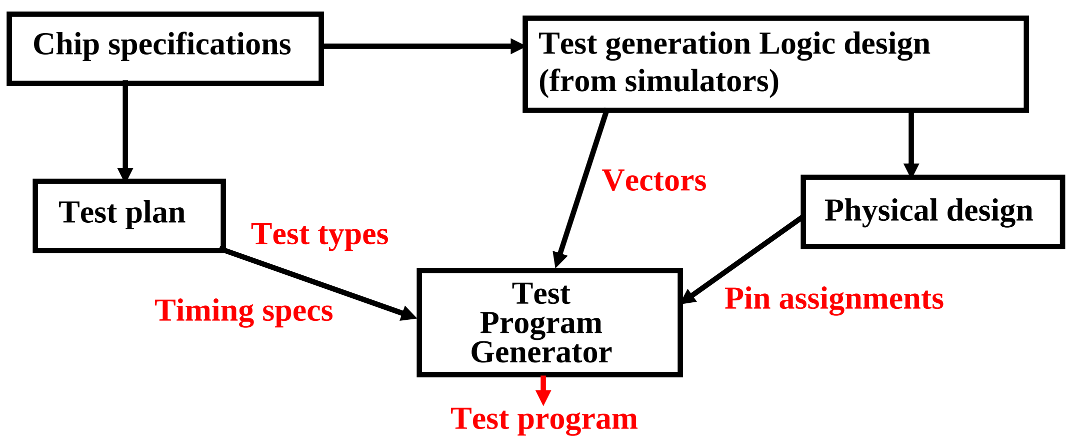

Fig. 38.3 Test program generation

6.3 Test Pattern Generation

Test pattern generation is the process of generating a (minimal) set of input patterns to stimulate the inputs of a circuit, such that detectable faults can be sensitized and their effects can be propagated to the output. The process can be done in two phases: (1) derivation of a test, and (2) application of a test. For (1), appropriate models for the circuit (gate or transistor level) and faults are to be decided. Construction of the test is to be accomplished in a manner such that the output signal from a faulty circuit is different from that of a good circuit. This can be computationally very expensive, but the task is to be performed offline and only once at the end of the design stage. The generation of a test set can be obtained either by algorithmic methods (with or without heuristics), or by pseudo-random methods. On the other hand, for (2), a test is subsequently applied many times to each integrated circuit and thus must be efficient both in space (storage requirements for the patterns) and in time. The main considerations in evaluating a test set are: (i) the time to construct a minimal test set; (ii) the size of the test set; (iii) the time involved to carry out the test; and (iv) the equipment required (if external). Most algorithmic test pattern generators are based on the concept of sensitized paths.

The Sensitized Path Method is a heuristic approach to generating tests for general combinational logic networks. The circuit is assumed to have only a single fault in it. The sensitized path method consists of two parts:

1. The creation of a SENSITIZED PATH from the fault to the primary output. This involves assigning logic values to the gate inputs in the path from the fault site to a primary output, such that the fault effect is propagated to the output.

2. The JUSTIFICATION operation, where the assignments made to gate inputs on the sensitized path is traced back to the primary inputs. This may require several backtracks and iterations.

In the case of sequential circuits the same logic is applied but before that the sequential elements are explicitly driven to a required state using scan based design-for-test (DFT) circuitry [1,24].

The best-known algorithms are the D-algorithm, PODEM and FAN [1,24]. Three steps can be identified in most automatic test pattern generation (ATPG) programs: (a) listing the signals on the inputs of a gate controlling the line on which a fault should be detected; (b) determining the primary input conditions necessary to obtain these signals (back propagation) and sensitizing the path to the primary outputs such that the signals and faults can be observed; (c) repeating this procedure until all detectable faults in a given fault set have been covered.

6.4 ATPG for Hardware-Software Covalidation

Several automatic test generation (ATG) approaches have been developed which vary in the class of search algorithm used, the fault model assumed, the search space technique used, and the design abstraction level used. In order to perform test generation for the entire system, both hardware and software component behaviors must be described in a uniform manner. Although many behavioral formats are possible, ATG approaches have focused on CDFG and FSM behavioral models.

Two classes of search algorithms have been explored, *fault directed* and *coverage directed*. Fault directed techniques successively target a specific fault and construct a test sequence to detect that fault. Each new test sequence is merged with the current test sequence (typically through concatenation) and the resulting fault coverage is evaluated to determine if test generation is complete. Fault directed algorithms have the advantage that they are complete in the sense that a test sequence will be found for a fault if a test sequence exists, assuming that sufficient CPU time is allowed. For test generation, each CDFG path can be associated with a set of constraints which must be satisfied to traverse the path. Because the operations found in a hardware-software description can be either boolean or arithmetic, the solution method chosen must be able to handle both types of operations. Constraint logic programming (CLP) techniques [27] are capable to handle a broad range of constraints including non-linear constraints on both boolean and arithmetic variables. State machine testing has been accomplished by defining a *transition tour* which is a path which traverses each state machine transition at least once 26ransition tours have been generated by iteratively improving an existing partial tour by concatenating on to it the shortest path to an uncovered transition [26 A significant limitation to state machine test generation techniques is the time complexity of the state enumeration process performed during test generation.

Coverage directed algorithms seek to improve coverage without targeting any specific fault. These algorithms heuristically modify an existing test set to improve total coverage, and then evaluate the fault coverage produced by the modified test set. If the modified test set corresponds to an improvement in fault coverage then the modification is accepted. Otherwise the modification is either rejected or another heuristic is used to determine the acceptability of the modification. The modification method is typically either random or directed random. An example of such a technique is presented in [25] which uses a genetic algorithm to successively improve the population of test sequences.

7. Embedded Software Testing

7.1 Software Unit Testing

The unit module is either an isolated function or a class. This is done by the development team, typically the developer and is done usually in the peer review mode. Test data /test cases are developed based on the specification of the module. The test case consists of either:

- *Data-intensive testing*: applying a large range of data variation for function parameter values, or
- *Scenario-based testing*: exercising different method invocation sequences to perform all possible use cases as found in the requirements.

Points of Observation are returned value parameters, object property assessments, and source code coverage. Since it is not easy to track down trivial errors in a complex embedded system, every effort should be made to locate and remove them at the unit-test level.

7.2 Software Integration Testing

All the unit modules are integrated together. Now the module to be tested is a set of functions or a cluster of classes**.** The essence of integration testing is the validation of the interface. The same type of Points of Control applies as for unit testing (data-intensive main function call or method-invocation sequences), while Points of Observation focus on interactions between lower-level models using information flow diagrams.
First, performance tests can be run that should provide a good indication about the validity of the architecture. As for functional testing, the earlier is the better. Each forthcoming step will then include performance testing. White-box testing is also the method used during that step. Therefore software integration testing is the responsibility of the developer.

7.3 Software Validation Testing

This can be considered one of the activities that occur toward the end of each software integration. Partial use-case instances, which also called partial scenarios, begin to drive the test implementation. The test implementation is less aware of and influenced by the implementation details of the module. Points of Observation include resource usage evaluation since the module is a significant part of the overall system. This is considered as white-box testing. Therefore, software validation testing is also the responsibility of the developer.

7.4 System Unit Testing

Now the module to be tested is a full system that consists of user code as tested during software validation testing plus all real-time operating system (RTOS) and platform-related pieces such as tasking mechanisms, communications, interrupts, and so on. The Point of Control protocol is no longer a call to a function or a method invocation, but rather a message sent/received using the RTOS message queues, for example. Test scripts usually bring the module under test into the desired initial state; then generate ordered sequences of samples of messages; and validate messages received by comparing (1) message content against expected messages and (2) date of reception against timing constraints. The test script is distributed and deployed over the various virtual testers. System resources are monitored to assess the system's ability to sustain embedded system execution. For this aspect, grey-box testing is the preferred testing method. In most cases, only a knowledge of the interface to the module is required to implement and execute appropriate tests. Depending on the organization, system unit testing is either the responsibility of the developer or of a dedicated system integration team.

7.5 System Integration Testing

The module to be tested starts from a set of components within a single node and eventually encompasses all system nodes up to a set of distributed nodes. The Points of Control and Observations (PCOs) are a mix of RTOS and network-related communication protocols, such as RTOS events and network messages. In addition to a component, a Virtual Tester can also play the role of a node. As for software integration, the focus is on validating the various interfaces. Grey-box testing is the preferred testing method. System integration testing is typically the responsibility of the system integration team.

7.6 System Validation Testing

The module to be tested is now a complete implementation subsystem or the complete embedded system. The objectives of this final aspect are several:

- *Meet external-actor functional requirements*. Note that an external-actor might either be a device in a telecom network (say if our embedded system is an Internet Router), or a person (if the system is a consumer device), or both (an Internet Router that can be administered by an end user).
- *Perform final non-functional testing* such as load and robustness testing. Virtual testers can be duplicated to simulate load, and be programmed to generate failures in the system.
- *Ensure interoperability with other connected equipment*. Check conformance to applicable interconnection standards. Going into details for these objectives is not in the scope of this article. Black-box testing is the preferred method: The tester typically concentrates on both frequently used and potentially risky or dangerous use-case instances.

8. Interaction Testing Technique between Hardware and Software in Embedded Systems

In embedded system where hardware and software are combined, unexpected situation can occur owing to the interaction faults between hardware and software. As the functions of embedded system get more complicated, it gets more difficult to detect faults that cause such troubles. Hence, Faults Injection Technique is strongly recommended in a way it observes system behaviors by injecting faults into target system so as to detect interaction faults between hardware and software in embedded system.

The test data selection technique discussed in [21] first simulates behaviors of embedded system to software program from requirement specification. Then hardware faults, after being converted to software faults, are injected into the simulated program. And finally, effective test data are selected to detect faults caused by the interactions between hardware and software.

9. Conclusion

Rapid advances in test development techniques are needed to reduce the test cost of million-gate SOC devices. In this chapter a number of state-of-the-art techniques are discussed for testing of embedded systems. Modular test techniques for digital, mixed-signal, and hierarchical SOCs must develop further to keep pace with design complexity and integration density. The test data bandwidth needs for analog cores are significantly different than that for digital cores, therefore unified top-level testing of mixed-signal SOCs remains major challenge. This chapter also described granular based embedded software testing technique.

References

[1] M. L. Bushnell and V. D Agarwal, "Essentials of Electronic Testing" Kluwer academic Publishers, Norwell, MA, 2000.

[2] E. A. Lee, "What's Ahead for Embedded Software?", IEEE Computer, pp 18-26, September, 2000.

[3] E. A. Lee, "Computing for embedded systems", proceeding of IEEE Instrumentation and Measurement Technology Conference, Budapest, Hungary, May, 2001.

[4] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors, 2001 Edition", http://public.itrs.net/Files/2001ITRS/Home.html

[5] Y. Zorian, E.J.Marinissen, and S.Dey, "Testing Embedded-Core Based System Chips", IEEE Computer, 32,52-60,1999

[6] M-C Hsueh, T. K.Tsai, and R. K. Lyer, "Fault Injection Techniques and Tools", IEEE Computer, pp75-82, April,1997.

[7] V. Encontre, "Testing Embedded Systems: Do You Have The GuTs for It?" www-128.ibm.com/developerworks/rational/library/content/03July/1000/1050/1050.pdf

[8] D. D. Gajski and F. Vahid, "Specification and design of embedded hardware-software systems", IEEE Design and Test of Computers, vol. 12, pp. 53–67, 1995.

[9] S. Dey, A. Raghunathan, and K. D. Wagner, "Design for testability techniques at the behavioral and register-transfer level", Journal of Electronic Testing: Theory and Applications (JETTA), vol. 13, pp. 79–91, October 1998.

[10] B. Beizer, Software Testing Techniques, Second Edition, Van Nostrand Reinhold, 1990.

[11] G. Al Hayek and C. Robach, “From specification validation to hardware testing: A unified method”, in International Test Conference, pp. 885–893, October 1996.
[12] A. von Mayrhauser, T. Chen, J. Kok, C. Anderson, A. Read, and A. Hajjar, “On choosing test criteria for behavioral level harware design verification”, in High Level Design Validation and Test Workshop, pp. 124–130, 2000.
[13] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, “A formal evaluation of data flow path selection criteria”, IEEE Trans. on Software Engineering, vol. SE-15, pp. 1318–1332, 1989.
[14] S. C. Ntafos, “A comparison of some structural testing strategies”, IEEE Trans. on Software Engineering, vol. SE-14, pp. 868–874, 1988.
[15] J. Laski and B. Korel, “A data flow oriented program testing strategy”, IEEE Trans. on Software Engineering, vol. SE-9, pp. 33–43, 1983.
[16] Q. Zhang and I. G. Harris, “A domain coverage metric for the validation of behavioral vhdl descriptions”, in International Test Conference, October 2000.
[17] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, “Abstraction techniques for validation coverage analysis and test generation”, IEEE Transactions on Computers, vol. 47, pp. 2–14, January 1998.
[18] N. Malik, S. Roberts, A. Pita, and R. Dobson, “Automaton: an autonomous coverage-based multiprocessor system verification environment”, in IEEE International Workshop on Rapid System Prototyping, pp. 168–172, June 1997.
[19] K.-T. Cheng and A. S. Krishnakumar, “Automatic functional test bench generation using the extended finite state machine model”, in Design Automation Conference, pp. 1–6, 1993.
[20] J. P. Bergmann and M. A. Horowitz, “Improving coverage analysis and test generation for large designs”, in International Conference on Computer-Aided Design, pp. 580–583, 1999.
[21] A. Sung and B. Choi, “An Interaction Testing Technique between Hardware and Software in Embedded Systems”, Proceedings of Ninth Asia-Pacific Software Engineering Conference, 2002. 4-6 Dec. 2002 Page(s):457 – 464
[22] IEEE P I500 Web Site. http://grouper.ieee.org/groups/I SOO/.
[23] H. Al-Asaad, B. T. Murray, and J. P. Hayes, “Online BIST for embedded systems” IEEE Design & Test of Computers, Volume 15, Issue 4, Oct.-Dec. 1998 Page(s): 17 – 24
[24] M. Abramovici, M.A. Breuer, AND A.D. Friedman, “Digital Systems Testing and Testable Design”, *IEEE Press* 1990.
[25] F. Corno, M. Sonze Reorda, G. Squillero, A. Manzone, and A. Pincetti, “Automatic test bench generation for validation of RT-level descriptions: an industrial experience”, *in Design Automation and Test in Europe*, pp. 385–389, 2000.
[26] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, “Architecture validation for processors”, *in International ymposium on Computer Architecture*, pp. 404–413, 1995.
[27] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.

Problems

1. How testing differs from verification?
2. What is embedded system? Define hard real-time system and soft real-time system with example.
3. Why testing embedded system is difficult?
4. How hardware testing differs from software testing?

5. What is co-testing?
6. Distinguish between defects, errors and faults with example.
7. Calculate the total number of single and multiple stuck-at faults for a logic circuit with n lines.
8. In the circuit shown in Figure 38.4 if any of the following tests detect the fault x_1 *s-a-0*?
   a) (0,1,1,1)
   b) (1,0,1,1)
   c) (1,1,0,1)
   d) (1,0,1,0)

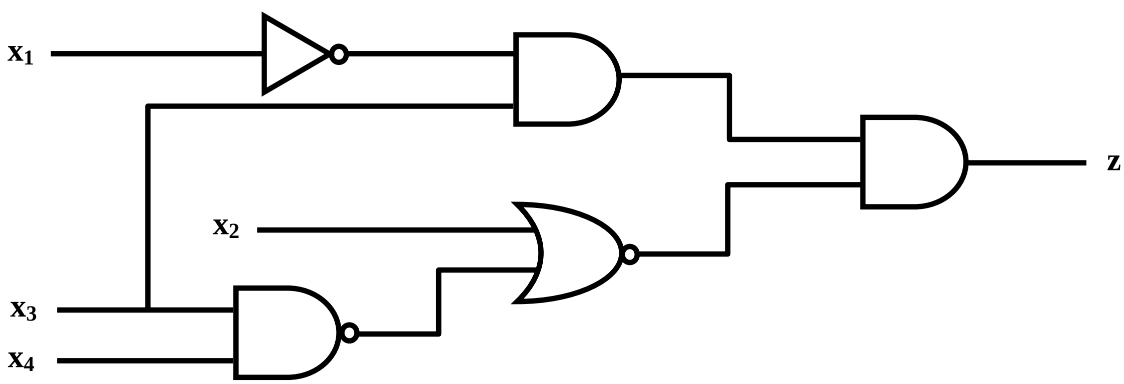

Fig. P1

9. Define the following fault models using examples where possible:
   a) Single and multiple stuck-at fault
   b) Bridging fault
   c) Stuck-open and stuck-short fault
   d) Operational fault
10. What is meant by co-validation fault model?
11. Describe different software fault model?
12. Describe the basic structure of core-based testing approach for embedded system.
13. What is concurrent or on-line testing? How it differs from non-concurrent testing?
14. Define error coverage, error latency, space redundancy and time redundancy in view of on-line testing?
15. What is a test vector? How test vectors are generated? Describe different techniques for test pattern generation.
16. Define the following for software testing:
    a) Software unit testing
    b) Software integration testing
    c) Software validation testing
    d) System unit testing
    e) System integration testing
    f) System validation testing