Chapter 8
Test

Unfortunately, we cannot rely on designed and possibly already manufactured systems to operate as expected. These systems may have become defective during their use, or their function may have been compromised during the fabrication or their design. The purpose of testing is to verify whether or not an existing embedded/cyber-physical system can be operated as expected. In this chapter, we will present fundamental terms and techniques for testing. There will be a brief introduction to the aims of test pattern generation and their application. We will be introducing terms such as fault model, fault coverage, fault simulation, and fault injection. Also, we will be presenting techniques which improve testability, including the generation of pseudo-random patterns, and signature analysis. It would be beneficial to consider testability issues already during design. In case of fault-tolerant systems, resilience must be verified.

8.1 Scope

Testing can be done during or after the fabrication (manufacturing test) and also after the system has been delivered to the customer (field testing). Testing of embedded systems contained in a cyber-physical or IoT system needs special attention for several reasons:

- Embedded systems integrated into a physical environment may be safety-critical. Therefore, their malfunctioning can be much more dangerous than, say, the malfunctioning of office equipment. As a result, expectations for the product quality are higher than for non-safety-critical systems.
- Testing of timing-critical systems has to validate the correct timing behavior. This means that just testing the functional behavior is not sufficient.

P. Marwedel, *Embedded System Design*, Embedded Systems,
https://doi.org/10.1007/978-3-030-60910-8_8

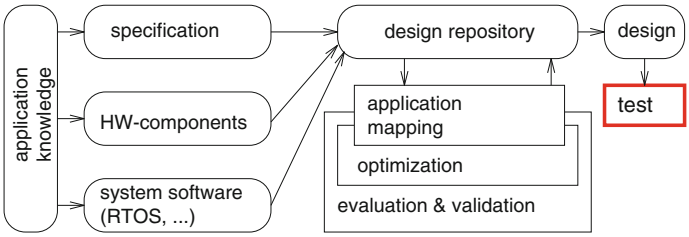


Fig. 8.1 Design flow with testing at its very end

- Testing embedded/cyber-physical systems in their real environment may be dangerous. For example, testing control software in a nuclear power plant can be a source of serious, far-reaching problems.

Preparations for testing should be done no later than at the end of the design phase. Preferably, necessary support for testing should even be considered earlier, intertwined with the design process and using testability as one of the objectives for evaluating designs. In order not to overload Chap. 5, we have moved all aspects of testing into this separate chapter. The presentation corresponds to considering testing only at the very end of the design flow (see Fig. 8.1), even though an earlier consideration during an actual design would be advisable. However, an early consideration is not always common practice, and therefore, Fig. 8.1 might also correspond to an actual design flow.

In testing, we are typically denoting the system under design (SUD) as the **device under test** (DUT). We are applying a set of specially selected input patterns, the so-called **test patterns** to the input(s) of the DUT, observe its behavior, and compare the behavior with the expected behavior. Test patterns are normally applied to the real, already manufactured system. The main purpose of testing is to identify systems that have not been correctly manufactured (manufacturing test) and to identify systems that fail later (field test). Testing includes a number of different actions:

1. **test pattern generation,**
2. **test pattern application,**
3. **response observation,** and
4. **result comparison**.

8.2 Test Procedures

8.2.1 Test Pattern Generation for Gate-Level Models

In test pattern generation, we try to identify a set of test patterns which distinguishes a correctly working from an incorrectly working system. Test pattern generation is

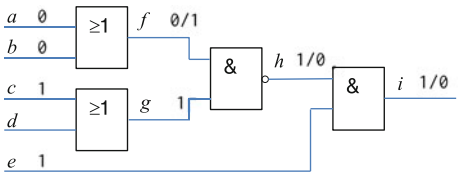


Fig. 8.2 Test pattern at the gate level

usually based on **fault models**. Such fault models are models of possible faults. Test pattern generation tries to generate tests for all faults that are possible according to a certain fault model.

The stuck-at-fault model is a frequently used fault model. It is based on the assumption that any internal wire of an electronic circuit is permanently connected to either `'0'` or `'1'`. It has been observed that many faults actually behave as if some wire was permanently connected that way.

Example 8.1 As an example, consider the circuit shown in Fig. 8.2.[1]

Suppose that we would like to check if there is a stuck-at-1 fault for signal f. Toward this end, we try to set f to `'0'` by setting $a = b =$`'0'`. As a result, f should be `'1'` if there is this fault, and otherwise, it should be `'0'`. In order to observe this difference, we must propagate it to the output signal i. For this to happen, we must set e to `'1'` and set either c or d to `'1'`. h and i will be `'1'` if there is no fault and `'0'` otherwise. The test pattern comprises all values of inputs a to e. The D-algorithm can be used to generate this test pattern [318]. ∇

Many techniques for test pattern generation are based on the stuck-at-fault model. However, CMOS technologies require more comprehensive fault models. In CMOS technologies, faults can turn combinatorial devices into devices having internal states. This problem can occur if wires are broken (this case is known as **stuck-at-open fault**). As a result of this, gates of transistors can become disconnected. Such transistors will be conducting or nonconducting, depending on the charge stored on the gate before the wire was broken. In this way, the gate "remembers" the input signal due to stored charges. Furthermore, there may be transient faults and delay faults (faults changing the delay of a circuit). Delay faults may be the result of cross talk between adjacent wires. Fault models exist which take such hardware faults into account [311].

While good fault models exist for hardware testing, the same is not true for software testing.

[1] Please remember: consistent with standard ANSI/IEEE 91, the symbols ≥1 and & denote OR- and AND-gates, respectively.

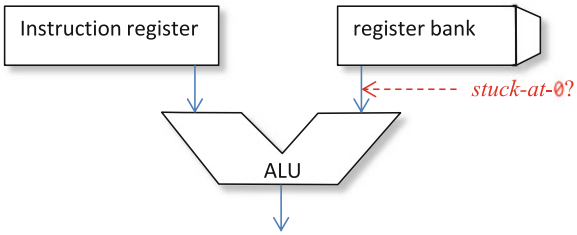


Fig. 8.3 Processor hardware components

8.2.2 Self-Test Programs

One of the key problems of testing modern integrated circuits is their limited number of pins, making it more and more difficult to access internal components. Also, it is getting very difficult to test these circuits at full speed, since testers must be at least as fast as the circuits themselves. The fact that many embedded systems are based on processors provides a way out of this dilemma: processors are capable of running test programs or *diagnostics*. Such diagnostics have been used to test main frame machines for decades.

Example 8.2 Figure 8.3 shows components that might be contained in a processor.

Testing for stuck-at-faults at the input of the ALU is feasible with a small test program:

```
store pattern of all '1's in the register file;
perform xor between constant "0000. . . 00" and register;
test if result contains a '0' bit;
if yes, report error;
otherwise start test for next fault;
```

∇

Similar small programs can be generated for other faults. Unfortunately, the process of generating diagnostics for main frames has mostly been a manual one. Some researchers have proposed to generate diagnostics automatically [48, 53, 64, 308, 312, 313].

8.3 Evaluation of Test Pattern Sets and System Robustness

8.3.1 Fault Coverage

The quality of test pattern sets can be evaluated using **fault coverage** as a metric.

Definition 8.1 Fault coverage is the percentage of potential faults that can be found for a given test pattern set:

$$Coverage = \frac{\text{Number of detectable faults for a given test pattern set}}{\text{Number of faults possible due to the fault model}}$$

In practice, achieving a good product quality requires fault coverages in the area of at least 98–99%. The requirements may be higher for particular systems. Also, special fault models may be necessary for certain hardware components (e.g., for batteries).

In addition to achieving a high coverage, we must also achieve a high **correctness coverage**. This means that a fault-free system must be recognized as such. Otherwise, it would be possible to achieve a 100% coverage by classifying all systems as faulty. Note the link to the metrics in Sect. 5.3.3.

In order to increase the number of options that exists for system validation, it has been proposed to use test methods already during the design phase. For example, test pattern sets can be applied to software models of systems in order to check if two software models behave in the same way. More time-consuming formal methods need to be applied only to those cases in which the system passed this test-based equivalence check.

8.3.2 Fault Simulation

It is currently not feasible (and it will probably not be feasible) to completely predict the behavior of systems in the presence of faults or to analytically compute the coverage. Therefore, the behavior of systems in the presence of faults is frequently simulated. This type of simulation is called **fault simulation**. In fault simulation, system models are modified to reflect the behavior of the system in the presence of a certain fault. The goals of fault simulation include:

- to know the effect of a fault of the components at the system level (i.e., to check whether faults are redundant)
- to know whether or not mechanisms for improving fault tolerance actually help.

Definition 8.2 Faults are called **redundant** if they do not affect the observable behavior of the system.

Fault simulation requires the simulation of the system for all faults feasible for the fault model and also for a possibly large number of different input patterns. Accordingly, fault simulation is an extremely time-consuming process. Different techniques have been proposed to speed up fault simulation.

One such technique applies to fault simulation at the gate level. In this case, internal signals are single-bit signals. This fact enables the mapping of a signal to a single bit of some machine word of a simulating host machine. AND- and OR-machine instructions can then be used to simulate Boolean networks. However, only a single bit would be used per machine word. Efficiency is improved with parallel fault simulation.

Definition 8.3 Fault simulation is called **parallel fault simulation** if $n > 1$ different test patterns are simulated at the same time, where n is the length of a bit vector supported as a machine data type of the simulating processor.

The values of each of the n test patterns are mapped to a different bit position in the machine. Executing the same set of AND- and OR-instructions will then simulate the behavior of the Boolean network for n test patterns instead of for just one.

AVX instructions mentioned on p. 154 are very useful for this.

8.3.3 Fault Injection

Fault simulation may be too time-consuming for real systems. If actual systems are available, fault injection can be used instead. In fault injection, real existing systems are modified, and the overall effect on the system behavior is checked. Fault injection does not rely on fault models (even though they can be used). Hence, fault injection has the potential of generating faults that would not have been predicted by a fault model. We can distinguish between two types of fault injection:

- local faults within the system
- faults in the environment (behaviors which do not correspond to the specification). For example, we can check how the system behaves if it is operated outside the specified temperature or radiation ranges.

Several methods can be used for fault injection:

- fault injection at the hardware level: examples include pin manipulation and electromagnetic and nuclear radiation
- fault injection at the software level: examples include toggling some memory bits.

The quality of fault injection depends on the "probe effect": probing might have an impact on the behavior of the system. This impact should be as small as possible and essentially be negligible.

According to experiments reported by Kopetz [303], software-based fault injection was essentially as effective as hardware-based fault injection. Nuclear radiation was a noticeable exception in that it generated errors which were not generated with other methods.

8.4 Design for Testability

8.4.1 Motivation

Ideas for test pattern generation for Boolean circuits have been presented in Subsection 8.2.1. For circuits implementing state machines (automata), test pattern generation is more difficult. Verifying whether or not two finite state machines are equivalent may require complex input sequences [301].

Example 8.3 The state chart of Fig. 2.25 is shown again in Fig. 8.4 for convenience. Suppose that we would like to test the transition from state C to state D. This requires us to get into state C first, by applying an appropriate sequence of input patterns. Assuming that we start from the default state, we have to generate a sequence comprising signals g and h. Next, we must generate input event i and check if output y is generated. Also, we need to check if we reached state D. We could apply input signal j and check if output z is emitted. Still, we would not be sure that we actually had reached state D. There could be a fault resulting in the generation of z for a transition from a different state. This procedure is rather complicated, takes a lot of time, and is susceptible to interference with other errors. Nevertheless, the procedure could even be more complicated since the overall test in our example is simplified by the fact that the FSM contains a linear chain of transitions (see the assignments of this chapter). ∇

This example demonstrates if testing comes in only as an afterthought, it may be very difficult to test a system. In order to simplify tests, special hardware can be added such that testing becomes easier. The process of designing for better testability is called **design for testability** (**DfT**). Special purpose hardware for testing finite state machines is a prominent example of this.

8.4.2 Scan Design

Reaching certain states and observing states resulting from the application of input patterns are very much simplified with **scan design**. In scan design, all flip-flops storing states are connected to form serial shift registers (see Fig. 8.5). The circuit

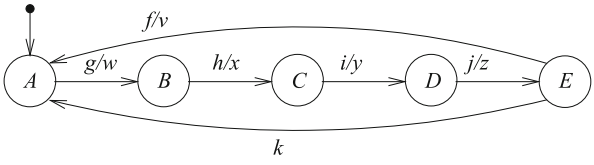
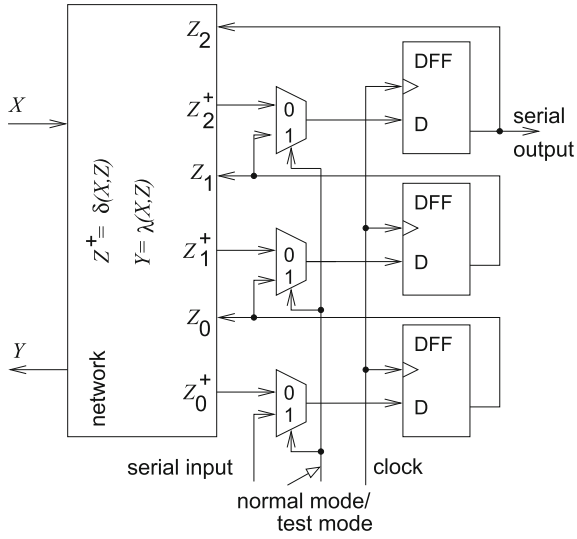


Fig. 8.4 Finite state machine to be tested

Fig. 8.5 Scan path design



contains three D-type flip-flops (DFF) and one multiplexer at each of the flip-flop inputs. Using the control input of the multiplexers (shown at the bottom of the multiplexer inputs), either we can connect the flip-flops to the network generating the next state from the current state and the current input or we can connect flip-flops to form a serial chain. Setting the multiplexers to scan mode, we can load state bit after state bit into the scan chain (1 bit at every clock tick). This way, we can load any state into the three flip-flops serially. In a second phase, we can apply an input pattern to the FSM while the multiplexers are set to normal mode. After the next clock tick, the FSM will be in a new state. This new state can be serially shifted out in the third and final phase, using the serial mode again (1 bit per clock tick). The net effect is that we do not need to worry about how to get into certain states and how to observe whether or not the Boolean function δ for computing the next state has been correctly implemented while we are generating tests for the FSM. Effectively, the fact that we are dealing with state-based systems has an impact only on the two (simple) shift phases, and test pattern generation for (stateless) Boolean networks can be used for checking for correct outputs. This means that it is sufficient to use test pattern generation methods for Boolean functions (stateless networks) instead of caring about complex input sequences, etc.

Scan design is a technique which works well for single chips. For board-level integration, it is necessary to have some technique for connecting scan chains of several chips. **JTAG** is a standard designed for this. The standard defines registers at the boundaries of all chips and a number of test pins and control commands such that all chips can be connected in scan chains. JTAG is also known as boundary scan [447].

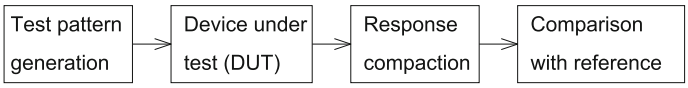


Fig. 8.6 Testing a device under test (DUT)

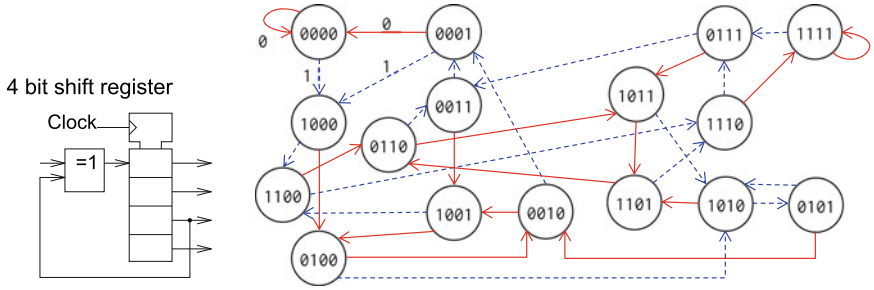


Fig. 8.7 LFSR for response compaction: **left**, schematic; **right**, state diagram

8.4.3 Signature Analysis

In order to also avoid shifting out the response of the device under test (DUT), responses can be compacted. A setup like the pipeline shown in Fig. 8.6 can be used for this purpose. Generated test patterns are used as inputs (or so-called stimuli) to the DUT. The response of the DUT is then compacted to form a **signature**, which characterizes the response. This response is later compared to the expected response. The expected response can be computed by simulation.

The compaction is usually performed with linear feedback shift registers (LFSRs), shift registers with an XOR-feedback.

Example 8.4 Figure 8.7 shows a 4-bit LFSR (left) and the associated state diagram (right) [318]. Blue dashed lines denote an input of `'1'`; red solid lines denote an input of `'0'`. The selected feedback yields all possible signatures. During testing, the response of the system tested is sent to the input of the LFSR. The LFSR will then generate a signature reflecting the response. ∇

Due to storing the signature instead of the full response, several response patterns can be mapped to the same signature. What is the probability of obtaining a correct signature from an incorrect response?

In general, an n-bit signature generator can generate 2^n signatures. For an m-bit response of the DUT, the best that we can do is to evenly map $2^{(m-n)}$ responses to the same signature. Suppose that we expect a certain signature to be generated for the correct response of the system. Then, $2^{(m-n)} - 1$ incorrect responses would also map to the same signature. There is a total of $2^m - 1$ incorrect responses if responses are m-bit long. Hence, the probability of an incorrect response to map to the correct signature (provided patterns map evenly to signatures) is

$$P = Pr\left(\frac{\text{other patterns mapping to the same signature}}{\text{total number of other patterns}}\right) \tag{8.1}$$

$$= \frac{2^{(m-n)} - 1}{2^m - 1} \tag{8.2}$$

$$\approx \frac{2^{(m-n)}}{2^m} \text{ for } m \gg n \tag{8.3}$$

$$\approx \frac{1}{2^n} \text{ for } m \gg n \tag{8.4}$$

This means that the probability of generating correct signatures from an incorrect test response is very small if the shift register is long. For example, actual shift registers may be 32 bits long. Nevertheless, it is still feasible to have the correct signature for wrong inputs. The corresponding effect is called **aliasing**. A careful analysis of aliasing is recommended at least for critical applications.

8.4.4 Pseudo-random Test Pattern Generation

For chips with a large number of flip-flops, it can take quite some time to shift in the test patterns. In order to speed up the process of generating patterns on the chip, it has been proposed to also integrate hardware for generating test patterns on the chip. This is especially useful when the bandwidth for accesses from outside the chip is much less than the internal bandwidth on the chip.

For example, pseudo-random patterns (also generated by LFSRs) can be used as test patterns. This method typically requires less chip space than patterns stored in a table.

Example 8.5 We can modify the circuit of Fig. 8.7 as shown in Fig. 8.8. The circuit generates all possible test patterns, except the pattern consisting of all zeros.

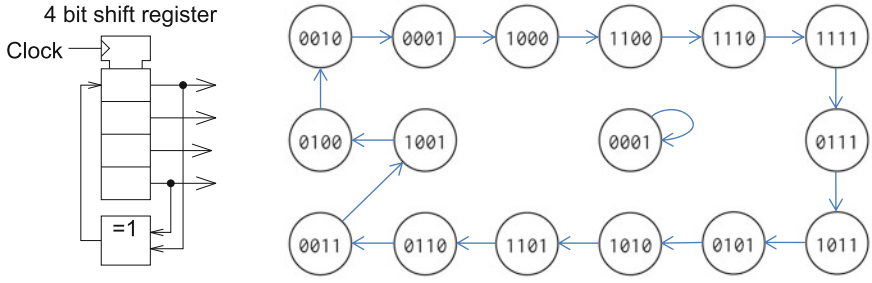
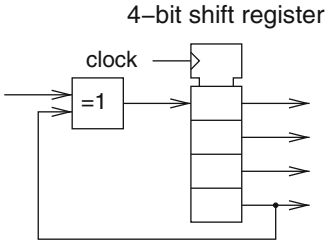


Fig. 8.8 Linear feedback shift register for test pattern generation ∇

Fig. 8.9 LFSR



Patterns consisting of all zeros have to be avoided, since the generator would get stuck once it arrives at such a pattern. The generated patterns are typically exercising systems to be tested much better than simple counters.

8.5 Problems

8.1 Consider the circuit shown in Fig. 8.2. Generate a test pattern for a stuck-at-`0` fault at signal h!

8.2 Which state diagram corresponds to the LFSR shown in Fig. 8.9?

8.3 Specify test patterns and expected responses for the FSM shown in Fig. 8.4. These patterns must be specified as a sequence of pairs (test pattern, expected response). Events shown in Fig. 8.4 can be used as test patterns. We assume that the FSM will be in the default state after power on. Provide a complete test for all transitions! Note that the special chain-like structure of the FSM simplifies testing.