Chapter 12
Validation

Overview This chapter deals with assessment technologies. These technologies must convince a designer, user, or a certification authority that the developed computer system is safe to deploy and will fulfill its intended function in the planned real-world environment. In Sect. 12.1 we elaborate on the differences between *validation* and *verification*. Validation deals with the consistency between the informal *model of the user's intention* and the behavior of the *system-under-test* (*SUT*), while verification deals with the consistency between a given (*formal*) *specification* and the *SUT*. The missing link between validation and verification are errors in the specification. The following section deals with the challenges of testing, the preferred validation technique. At the core of testing are the interference-free observability of results and the controllability of the inputs. The design for testability provides a framework that supports these characteristics. In most cases, only a tiny fraction of the input space can be examined by test cases. The proper selection of test cases should justify the assumption that, given the results of the test cases are correct, the system will operate correctly all over the input domain. In digital systems the validity of such an induction is doubtful, since digital inputs are not continuous but discrete – a single bit-flip can make a correct result erroneous. The decision whether the result of a test input is correct is delegated to a *test oracle*. The automation of test oracles is another challenge in the domain of testing. Model-based design, where a model of the plant and a model of the computer controller are interconnected to study the performance of closed-loop control systems is a promising route towards the automation of the test oracle. Given that a complete formal model of a design is available, formal methods can be deployed to check whether selected properties hold in all possible states of the model. In the last few years, the technique of model checking has matured such that it can handle systems of industrial size. The correct operation of the fault-masking mechanisms of a fault-tolerant system can only be assessed if the input space is extended to include the faults the system is supposed to tolerate. In the last section, the topics of physical fault-injection and software-based fault injection are covered. Since any physical sensor or actuator will eventually fail, fault-injection campaigns must establish the safe operation of a system even in the case that any particular sensor or actuator has failed.

12.1 Validation Versus Verification

An essential fraction – up to 50% – of the development costs of a real-time computer system is devoted to ensure that the system is *fit-for-purpose*. In safety-critical applications that must be certified, this fraction is even higher.

When developing an embedded computer system, it is useful to distinguish between three different types of system representations [Gau05]:

1. The informal *model of the user's intention* that determines the role of the embedded computer system in the given real-world application context. In the embedded world, this model deals with the relationships of the computer inputs and outputs to the effects in the physical environment. This model is usually not fully documented and informal, since in most cases it is not possible to think about and formalize all system aspects that are relevant in a real-world scenario.
2. The model of the system specification which captures and documents, either in natural language or in some formal notation, the intentions of the client and the obligations of the system developers as understood by the person or the group of persons who develop the (formal) specification.
3. The system under test (SUT) (the result of the system development) that should perform the system functions according to the model of the user's intention.

Verification establishes the consistency between a (*formal*) *system specification* and the *SUT*, while *validation* is concerned with the consistency between the *model of the user's intention* and the *SUT*. The missing link between verification and validation is the relation between the (*informal*) *model of the user's intention* and the (*formal*) *specification* of the system. We call errors that occur in this phase of development *specification errors*, while we call errors that occur during the transformation of a given specification to the SUT *implementation errors*. While *verification* can, in theory, be reduced to a formal process, *validation* must examine the system's behavior in the real world. If properties of a system have been formally verified, it still has not been established whether the existing formal specification captures all aspects of the intended behavior in the user's environment, i.e., if it is free of *specification errors*. Sometimes the term *specification testing* is used to find out whether the specification is consistent with the model of the user's intentions [Gau05].

Validation, specification testing, and verification are thus three complementary means to support quality assurance. The prime validation method is *testing*, while the prime verification method is *formal analysis*.

During testing, the behavior of a real-time computer system is exercised at carefully selected points of the input domain and the corresponding results in the domains of value and time are classified as *correct* or *erroneous*. It is assumed, given that the test cases have been properly selected and correctly executed, that the *induction* that the program will operate correctly at all points of the enormous input space is justified. In a digital system, where the change of a single bit can have drastic consequences on the behavior, this induction is *fragile*. If we take a purely probabilistic point of view, an estimate that the *mean time to failure* (MTTF) of the SUT will be larger than a given *number of hours* can only be made if operational tests have been executed for a duration that corresponds to this *number of hours* [Lit93]. In practice, this means that it is not possible to establish an MTTF of more than 10^3–10^5 h by operational testing. This is orders of magnitude lower than the desired MTTF of safety-critical systems, which is in the order of 10^9 h.

The main shortcoming of formal methods is the missing link between the informal *model of the user's intention* and the *formal specification* that is the reference for assessing the correctness of the system. Furthermore, only a subset of the properties relevant for the system operation can be captured in *formal properties* that are examined during the formal analysis.

12.2 Testing Challenges

Observability of the outputs of the *SUT* and *controllability* of the test inputs are the core of any testing activity.

In non-real-time systems, the *observability* and *controllability* are provided by test- and debug monitors that halt the program flow at a test point and give the tester the opportunity to monitor and change program variables. In distributed real-time systems, such a procedure is not suitable for the following two reasons:

1. The temporal delay introduced at the test points modifies the temporal behavior of the system in such a manner that existing errors can be hidden and new errors can be introduced. This phenomenon is called the *probe effect*. Probe effects have to be avoided when testing real-time systems.
2. In a distributed system, there are many *loci* of control. The halting of one control path introduces a temporal distortion in the coordinated control flow that can lead to new errors.

12.2.1 Design for Testability

By *design for testability,* we mean the design of a framework and the provision of mechanisms that facilitate the testing of a system. The following techniques improve the testability:

1. Partitioning the system into composable subsystems with observable and in the domains of value and time well-specified interfaces. It is then possible to test each subsystem in isolation and to limit the integration effects to the testing of the emerging behavior. Probe-effect free observability of subsystem outputs at any level of the architecture was one of the motivations to provide multicasting in the basic message transport service (see Sect. 4.3) of the time-triggered architecture.

2. Establishment of a static temporal control structure such that the temporal control structure is independent of the input data. It is then possible to test the temporal control structure in isolation.
3. Reducing the temporal dimension of the input space by introducing a sparse time-base of proper granularity. The granularity of this time-base should be sufficient for the application at hand but should not be any smaller. The smaller the granularity of the sparse time-base, the larger the potential input space in the temporal domain. By decreasing the size of the input space or by increasing the number of non-redundant test cases the test coverage, i.e., the fraction of the total input space that is covered by the tests, can be improved.
4. Publication of the ground state of a node in a g-state message at the periodic reintegration point. The ground state can then be observed by an independent component without probe effect.
5. Provision of determinism in the software such that the same output messages will be produced if the same input messages are applied to a component.

Because of their deterministic properties and their static control structure, time-triggered systems are easier to test than event-triggered systems.

12.2.2 Test Data Selection

During the test phase, only a tiny fraction of the potential input space of a computer system can be exercised. The challenge for the tester is to find an effective and representative *test-data set* that will give the designer confidence that the system will work correctly for *all* inputs. In this section, we present some methods for test data selection:

Random Test Data Selection. Test data are selected randomly without any consideration of the program structure or the operational profile of use.

Requirements Coverage. In this method, the requirements specification is the starting point for selecting the test data. For each one of the given requirements, a set of test cases is designed to check whether the requirement is satisfied. The hidden assumption in this criterion is that the set of requirements is complete.

White-box Testing. The internal structure of a system is examined to derive a set of test data such that some kind of coverage criterion is satisfied, e.g., that all statements have been executed or that all branches of a program have been tested. This test data selection criterion is most effective for unit testing, where the internals of the component implementation are available.

Model-based Test Data Selection: The test data is derived from a model of the system under test and a model of the physical plant. Model-based test data selection can be automated, since the correctness of test results can be related to a performance criterion of the physical process.

Example: Consider the case where a *controller of an automotive engine* is tested versus a *model of this engine*. The model of the engine has been extensively validated with respect to the operation of the real engine and is assumed to be correct. The control algorithms that are implemented in the controller determine the performance parameters of the engine such as energy efficiency, torque, pollution, etc. By observing the performance parameters of the engine we can detect anomalies that are caused by a misbehavior of the controller software.

Operational profile. The basis of the test data selection is the operational profile of the *system under test* in the given application context. This test data selection criterion misses rare events.

Peak load. A hard real-time system must provide the specified timely service under all conditions covered by the load- and fault-hypothesis, i.e., also under peak loads that are caused by *rare events*. The peak-load scenario puts extreme stress on the system and should be tested extensively. The behavior of the system in above-peak-load situations must also be tested. *If peak load activity is handled correctly, the normal load case will take care of itself*. In most cases it is not possible to generate rare events and peak load in the real-world operational environment. Therefore peak-load testing is best performed in a model-based test environment.

Worst-Case Execution Time (*WCET*). To determine the WCET of a task experimentally, the task source code can be analyzed to generate a test data set that is biased towards the worst-case execution time.

Fault-Tolerance Mechanisms. Testing the correctness of the fault-tolerance mechanism is difficult, because faults are not part of the *normal* input domain. Mechanisms must be provided that can activate the faults during the test phase. For example, software- or hardware-implemented fault injection can be used to test the correctness of the fault-tolerance mechanisms (see Sect. 12.5).

Cyclic systems. If a system has a cyclic behavior (many control systems are cyclic), the crossing of a particular phase of the cycle is a repetitive event in the temporal domain. In many cyclic systems it is sufficient to test all events that occur in a single cycle.

The above list of test data selection criteria is not complete. A survey study on the effectiveness of different test data selection criteria is contained in Juristo et al. [Jur04]. Selecting the test data by using a combination of the above criteria seems to be more effective than relying on a single criterion in isolation.

In order to be able to judge the quality of a test-data set, different *coverage measures* have been introduced. A *coverage measure* describes the degree to which a test data set exercises the SUT. Common coverage criteria are:

1. Function coverage – has every function been exercised?
2. Statement coverage – has every statement of the source code been executed?
3. Branch coverage – has every branch instruction been executed in all directions?
4. Condition coverage – has every Boolean condition been fully exercised?
5. Fault coverage – have the fault-tolerant mechanisms been tested for every fault that is contained in the fault hypothesis?

12.2.3 Test Oracle

Given that a set of test cases has been selected, a method must be provided to determine whether the result of a test case produced by the SUT is acceptable or not. In the literature, the term *test oracle* is used to refer to such a method. The design of an *algorithmic test oracle* is a prerequisite of any test automation, which is needed for reducing the cost of testing.

In practice, the judgment whether the result of a test case is in conformance with a natural language representation of the *model of the user's intention* is often delegated to a human. Model-based design and model-based testing can help to partially solve the problem.

The structured design process, discussed in Sect. 11.2, distinguishes between the PIM (Platform-Independent-Model) and the PSM (Platform-Specific Model) of a component. An executable representation of the complete interface behavior (in the domains of value and time) at the PIM level of a design can act as the reference for the adjudication of a test result at the PSM level and thus help to detect *implementation errors*. The *oracle challenge* is thus shifted from the PSM level to the PIM level. Since the PIM is developed in an early phase of the design, errors can be captured early in the lifecycle, which reduces the cost associated with correction of the errors.

The LIF specification of a component (see Sect. 4.4.2) should contain *input assertions* and *output assertions*. Input assertions limit the input space of the component and exclude input data that the component is not designed to handle. Output assertions help to immediately detect errors that occur inside a component. Both input assertions and output assertions can be considered to act as a *test-oracle light* [Bar01]. Since the PIM is not resource constrained, the wide use of input assertions and output assertions at the PIM level can help to debug the PIM specification. In the second phase, when the PIM is transformed to the PSM, some of these assertions can be removed to arrive at an efficient code for the target machine.

In a time-triggered system, where the temporal control structure is static, the detection of temporal errors in the execution of a program at the PSM level is straightforward and can be automated.

12.2.4 System Evolution

Most successful systems evolve over time. Existing deficiencies are corrected and new functions are introduced in new versions of the system. The validation of these new versions must be concerned with two issues:

1. Regression *testing*. Checking that the functionality of the previous version (that must be supported in the new version) has not been modified and is still correct.
2. *New-function testing*. Checking that the functionality that is new to the latest version is implemented correctly.

Regression testing can be automated by executing the test-data set of the previous version on the new version. The anomalies that have been detected in the old version (see Sect. 6.3) can be the source of new test cases that put extraordinary stress on the system. Bertolino [Ber07] lists six questions that have to be addressed when designing a specific test campaign:

1. *WHY* do we perform the test? Is the objective of the test campaign to find residual design errors, to establish the reliability of a product before it can be released or to find out whether the system has a usable man–machine interface?
2. *HOW* to choose the test cases? There are different options to choose the test cases: random, guided by the operational profile, looking at a specific demand on the system (e.g., a shut down scenario of a nuclear reactor), or based on knowledge about the internal structure of the program (see Sect. 12.2.2).
3. *HOW MUCH* testing is sufficient? The decision about how many test cases must be executed can be derived from coverage analysis or from reliability considerations.
4. *WHAT* is it that we execute? What is the system under test – a module, the system in a simulated environment or the system in its real-world target environment?
5. *WHERE* do we perform the observation? This depends on the structure of the system and the possibility to observe the behavior of subsystems without disturbing the system operation.
6. *WHEN* is it in the product lifecycle that we perform the test? In the early phases of the lifecycle, testing can only be performed in an artificial laboratory environment. The real test comes when the system is exercised in its target environment.

12.3 Testing of Component-Based Systems

The component-based design of embedded applications, which is the focus of this book, requires appropriate strategies for the validation of component-based systems. A component is a hardware/software unit that encapsulates and hides its design and makes its services available at the message-based LIF (linking interface) of the component. While the component provider has knowledge about the internals of a component and can use this knowledge to arrive at effective test cases, the component user sees a component as a black box and must use and test the component on the basis of the given interface specification.

12.3.1 Component Provider

A component provider sees a component independent from the context of use. The provider must be concerned with the correct operation of the component in all possible user scenarios. In our component model, the user scenarios can be parameterized via the Technology Independent Interface (TII, see Sect. 4.4.3). The component provider must test the proper functioning of the component in the full parameter space supported by the TII. The component provider has access to the source code and can monitor the internal execution within the component across the Technology Dependent Interface (see Sect. 4.4.4), which is normally not utilized by the component user.

12.3.2 Component User

The *component user* is concerned with the performance of the component in its concrete *context of use* that is defined by a concrete parameter setting of the component for the given application. The component user can assume that the provider has tested the functions of the component as specified by the interface model and will put the focus of testing on the effects of component integration and the emerging behavior of a set of components, which is outside the scope of testing of the component provider.

In a first step, a component user must validate that the *prior properties of the component*, i.e., the properties that the component supplier has tested in isolation, are not refuted by the integration of the component. The component integration framework plays an important role in this phase.

In an event-triggered system, queues must be provided at the entry and exit of a component to align the component performance with the pending user requests, with the user's capability to absorb the results in a timely manner and with the transport capabilities of the communication system. Since every queue has a potential for overflow, flow-control mechanisms are needed across component boundaries. In the test phase the reaction of the component to queue overflow must be examined.

The integration of components can give rise to planned or unanticipated emergent behavior that is caused by the component interactions. *Emergent behavior is that which cannot be predicted through analysis at any level simpler than that of the system as a whole* [Dys98, p. 9]. This definition of emergent behavior makes it clear that the detection and handling of emergent behavior is in the realm of a component user and not of the component supplier. Mogul [Mog06] lists many examples of emergent behavior in computer systems that are caused by the interaction of components. The appearance of emergent behavior is not well-understood and a subject of current research (see also Sect. 2.4).

12.3.3 Communicating Components

During system integration, commercial-off-the-shelf (COTS) components or application-specific components are connected by their corresponding linking interfaces (LIFs). The message-exchange across these linking interfaces must be carefully tested. In Sect. 4.6 we have introduced three levels of a LIF specification: the *transport level*, the *operational level* and the *semantic level*. The LIF tests can follow along these three levels. The test at the transport level and the operational level, which must be precisely specified, can be performed mechanically, while the test of the meta-level (the semantics) will normally need human intervention. The multi-cast capability of the BMTS (basic message transport service) – see Sect. 6.1) enables the probe-effect-free observation of the information exchanged among communicating components.

In *model-based design* (Sect. 11.3.1), executable models of the behavior of the *physical plant* and of the *control algorithms for the computer system* are developed in parallel. At the PIM level these models, embodied in components, can be linked in a simulation environment such that the interaction of these components can be observed and studied. In a control system, the performance (quality of control) of the closed loop system can be monitored and used to find the optimal control parameter setting. The simulation will normally operate on a different time-scale than the target system. In order to improve the faithfulness of the simulation with respect to the target system, the phase relationships of the messages exchanged between the PIM components of the plant and of the controller should be the same as the phase relationships in the final implementation, the PSM. This constant phase relationship will avoid many subtle design errors caused by an uncontrolled and unintended phase relationship among messages [Per10].

12.4 Formal Methods

By the term *formal methods* we mean the use of mathematical and logical techniques to express, investigate, and analyze the specification, design, documentation, and behavior of computer hardware and software. In highly ambitious projects, formal methods are applied to *prove formally* that a piece of software implements the specification correctly. John Rushby [Rus93, p. 87] summarizes the benefits of formal methods as follows:

> Formal methods can provide important evidence for consideration in certification, but they can no more "prove" that an artifact of significant logical complexity is fit for its purpose than a finite-element calculation can "prove" that a wing span will do its job. Certification must consider multiple sources of evidence, and ultimately rests on informed engineering judgment and experience.

12.4.1 Formal Methods in the Real World

Any formal investigation of a real-world phenomenon requires the following steps to be taken:

1. *Conceptual model building*. This important informal first step leads to a precise natural language representation of the real-world phenomenon that is the subject of investigation.
2. *Model formalization*. In this second step, the natural language representation of the problem is transformed, and expressed in a formal specification language

with precise syntax and semantics. All assumptions, omissions, or misconceptions that are introduced in this step will remain in the model, and limit the validity of the conclusions derived from the model. Different degrees of rigor can be distinguished.

3. *Analysis of the formal model.* In the third step, the problem is formally analyzed. In computer systems, the analysis methods are based on discrete mathematics and logic. In other engineering disciplines, the analysis methods are based on different branches of mathematics, e.g., the use of differential equations to analyze a control problem.
4. *Interpretation of the results.* In the final step, the results of the analysis must be interpreted and applied to the real world.

Only step (3) out of these four steps can be fully mechanized. Steps (1), (2), and (4) will always require human involvement and human intuition and are thus as fallible as any other human activity.

An *ideal and complete* verification environment takes the *specification*, expressed in a formally defined specification language, the *implementation*, written in a formally defined implementation language, and the *parameters of the execution environment* as inputs, and establishes mechanically the *consistency* between specification and implementation. In a second step, it must be ensured that all assumptions and architectural mechanisms of the target machine (e.g., the properties and timing of the instruction set of the hardware) are consistent with the model of computation that is defined by the implementation language. Finally, the correctness of the verification environment itself must be established.

12.4.2 Classification of Formal Methods

Rushby [Rus93] classifies the use of formal methods in computer science according to the increasing rigor into the following three levels:

1. Use of concepts and notation of discrete mathematics. At this level, the sometimes ambiguous natural language statements about requirements and specification of a system are replaced by the symbols and conventions of discrete mathematics and logic, e.g., set theory, relations, and functions. The reasoning about the completeness and consistency of the specification follows a semi-formal manual style, as it is performed in many branches of mathematics.
2. Use of formalized specification languages with some mechanical support tools. At this level, a formal specification language with a fixed syntax is introduced that allows the mechanical analysis of some properties of the problems expressed in the specification language. At level (2), it is not possible to generate complete proofs mechanically.
3. Use of fully formalized specification languages with comprehensive support environments, including mechanized theorem proving or proof checking.

At this level, a precisely defined specification language with a direct interpretation in logic is supplied, and a set of support tools is provided to allow the mechanical analysis of specifications expressed in the formal specification language.

12.4.3 Benefits of Formal Methods

Level (1) *methods*. The compact mathematical notation introduced at this level forces the designer to clearly state the requirements and assumptions without the ambiguity of natural language. Since familiarity with the basic notions of set theory and logic is part of an engineering education, the disciplined use of level (1) methods will improve the communication within a project team and within an engineering organization and enrich the quality of documentation. Since most of the serious faults are introduced early in the lifecycle, the benefits of the level (1) methods are most pronounced at the early phases of requirements capture and architecture design. Rushby [Rus93, p. 39] sees the following benefits in using level (1) methods early in the lifecycle:

- The need for effective and precise communication between the software engineer and the engineers from other disciplines is greatest at an early stage, when the interdependencies between the mechanical control system and the computer system are specified.
- The familiar concepts of discrete mathematics (e.g., set, relation) provide a repertoire of mental building blocks that are precise, yet abstract. The use of a precise notation at the early stages of the project helps to avoid ambiguities and misunderstandings.
- Some simple mechanical analysis of the specification can lead to the detection of inconsistencies and omission faults, e.g., that symbols have not been defined or variables have not been initialized.
- The reviews at the early stages of the lifecycle are more effective if the requirements are expressed in a precise notation than if ambiguous natural language is used.
- The difficulty to express vague ideas and immature concepts in a semiformal notation helps to reveal problem domains that need further investigation.

Level (2) *methods*. Level (2) methods are a mixed blessing. They introduce a rigid formalism that is cumbersome to use, without offering the benefit of mechanical proof generation. Many of the specification languages that focus on the formal reasoning about the temporal properties of real-time programs are based at this level. Level (2) formal methods are an important intermediate step on the way to provide a fully automated verification environment. They are interesting from the point of view of research.

Level (3) *methods*. The full benefits of formal methods are only realized at this level. However, the available systems for verification are not complete in the sense that they cover the entire system from the high level specification to the hardware architecture. They introduce an intermediate level of abstraction that is above the functionality of the hardware. Nevertheless, the use of such a system for the rigorous analysis of some critical functions of a distributed real-time system, e.g., the correctness of the clock synchronization, can uncover subtle design faults and lead to valuable insights.

12.4.4 Model Checking

In the last few years, the verification technique of *model checking*, a level (3) method, has matured to the point that it can be used for the analysis and the support of the certification of safety-critical designs [Cla03]. Given a formal behavioral model of the specification and a formal property that must be satisfied, the *model checker* checks automatically whether the property holds in all states of the system model. In case the model checker finds a violation, it generates a concrete counter example. The main problem in model checking is the state explosion. In the last few years, clever formal analysis techniques have been developed to get a handle on the state explosion problem such that systems of industrial size can be verified by model checking.

12.5 Fault Injection

Fault injection is the intentional introduction of faults by software or hardware techniques in order to validate the system behavior under fault conditions. During a fault-injection experiment, the target system is exposed to two types of inputs: the *injected faults* and the *input data*. The faults can be seen as *another type of input* that activates the fault-management mechanisms. Careful testing and debugging of the fault-management mechanisms are necessary because a notable number of system failures is caused by errors in the fault-management mechanisms.

Fault injection serves two purposes during the evaluation of a dependable system:

1. *Testing and Debugging*. During normal operation, faults are *rare events* that occur only infrequently. Because a fault-tolerance mechanism requires the occurrence of a fault for its activation, it is very cumbersome to test and debug the operation of the fault-tolerance mechanisms without artificial fault injection.
2. *Dependability Forecasting*. This is used to get experimental data about the likely dependability of a fault-tolerant system. For this second purpose, the types and distribution of the expected faults in the envisioned operational environment must be known.

Table 12.1 Fault injection for testing and debugging versus dependability forecasting [Avr92]

	Testing and debugging	Dependability forecasting
Injected faults	Faults derived from the specified fault hypothesis	Faults expected in the operational environment
Input data	Targeted input data to activate the injected faults	Input data taken from the operational environment
Results	Information about the operation and effectiveness of the fault-tolerance mechanisms	Information about the envisioned dependability of the fault-tolerant system

Table 12.1 compares these two different purposes of fault injection.

It is possible to inject faults into the state of the computation (*software-implemented fault injection*) or at the physical level of the hardware (*physical fault injection*).

12.5.1 Software-Implemented Fault Injection

In *software-implemented fault injection,* errors are seeded into the memory of the computer by a fault-injection software tool. These seeded errors mimic the effects of hardware faults or design faults in the software. The errors can be seeded either randomly or according to some preset strategy to activate specific fault-handling tasks.

Software implemented fault injection has a number of potential advantages over physical fault injection:

1. Predictability: The space (memory cell) where and the instant when a fault is injected is fixed by the fault-injection tool. It is possible to reproduce every injected fault in the value domain and in the temporal domain.
2. Reachability: It is possible to reach the inner registers of large VLSI chips. Pin-level fault injection is limited to the external pins of a chip.
3. Less effort than physical fault injection: The experiments can be carried out with software tools without any need to modify the hardware.

12.5.2 Physical Fault Injection

During physical fault-injection, the target hardware is subjected to adverse physical phenomena that interfere with the correct operation of the computer hardware. In the following section, we describe a set of hardware fault-injections experiments that have been carried out on the MARS (Maintainable Real-time System) architecture in the context of the ESPRIT Research Project *Predictably Dependable Computing Systems* (*PDCS*) [Kar95,Arl03].

Table 12.2 Characteristics of different physical fault-injection techniques

Fault injection technique	Heavy-ion	Pin-level	EMI
Controllability, space	Low	High	Low
Controllability, time	None	High/medium	Low
Flexibility	Low	Medium	High
Reproducibility	Medium	High	Low
Physical reachability	High	Medium	Medium
Timing measurement	Medium	High	Low

The objective of the MARS fault-injection experiments was to determine the *error-detection coverage* of the MARS nodes experimentally. Two replica-determinate nodes receive identical inputs and should produce the same result. One of the nodes is subjected to fault-injections (the *FI-node*), the other node serves as a reference node (a *golden node*). As long as the consequences of the faults are detected within the FI-node, and the FI-node turns itself off, or the FI-node produces a detectably incorrect result message, the error has been classified as detected. If the FI-node produces a result message different from the result message of the golden node without any error indication, a fail-silence violation has been observed.

Three different fault-injection techniques were chosen at three different sites (see Table 12.2). At Chalmers University in Goeteborg, the CPU chip was bombarded with α particles until the system failed. At LAAS in Toulouse, the system was subjected to pin-level fault-injection, forcing an equi-potential line on the board into a defined state at a precise moment of time. At the Technische Universität Wien, the whole board was subjected to Electromagnetic Interference (EMI) radiation according to the IEC standard IEC 801-4.

Many different test runs, each one consisting of 2,000–10,000 experiments, were carried out with differing combinations of error detection techniques enabled. The results of the experiments can be summarized as follows:

1. With all error detection mechanisms enabled, no fail-silence violation was observed in any of the experiments.
2. The end-to-end error detection mechanisms and the double execution of tasks were needed in experiments with every one of the three fault-injection methods if error-detection coverage of >99% must be achieved.
3. In the experiment that used heavy-ion radiation, a triple execution was needed to eliminate all coverage violations. The triple execution consisted of a test-run with known outputs between the two replicated executions of the application task. This intermediate test run was not needed in the EMI experiments and the pin-level fault injection.
4. The bus guardian unit was needed in all three experiments if a coverage of >99% must be achieved. It eliminated the most critical failure of a node, the babbling idiots.

A detailed description of the MARS fault injection experiments and a comparison with software-fault injection carried out on the same system is contained in Arlat et al. [Arl03].

12.5.3 Sensor and Actuator Failures

The sensors and actuators, placed at the interface between the *physical world* and *cyberspace*, are physical devices that will eventually fail, just like any other physical device. The failures of sensors and actuators are normally not spontaneous *crash failures*, but manifest themselves either as transient malfunctions or a gradual drift away from the correct operation, often correlated with extreme physical conditions (e.g., temperature, vibration). An undetected sensor failure produces erroneous inputs to the computational tasks that, as a consequence, will lead to erroneous outputs that can be safety-relevant. Therefore it is state-of-the-art that any industrial-strength embedded system must have the capability to detect or mask the failure of any one of its sensors and actuators. This capability must be tested by fault-injection experiments, either software-based or physical.

An actuator is intended to transform a digital signal, generated in cyberspace, to some physical action in the environment. The incorrect operation of an actuator can only be observed and detected if one or more sensors observe the intended effect in the physical environment. This error-detection capability with respect to actuator failures must also be tested by fault-injection experiments (see also the example in Sect. 6.1.2).

In safety-critical applications, these fault-injection tests must be carefully documented, since they form a part of the safety case.

Points to Remember

- An essential fraction – up to 50% – of the development costs of a real-time computer system is devoted to ensure that the system is *fit-for-purpose*. In safety-critical applications that must be certified, this fraction is even higher.
- Verification establishes the consistency between a (formal) specification with the system under test (SUT), while validation is concerned with the consistency between the model of the user's intention with the SUT. The missing link between verification and validation is the relation between the model of the user's intention and the (formal) specification of the system.
- If a purely probabilistic point of view is taken, then an estimate that the mean time to failure (MTTF) of the SUT will be larger than a given number of hours can only be made if system tests have been executed for a duration that is larger than this number of hours.
- The modification of the behavior of the object under test by introducing a test probe is called the probe effect.
- Design for testability establishes a framework where test-outputs can be observed without a probe effect and where test inputs can be controlled at any level of the system architecture.
- It is a challenge for the tester to find an effective and representative set of test-data that will give the designer confidence that the system will work correctly for all inputs. A further challenge relates to finding an effective automatable test oracle.

- In the last few years clever formal techniques have been developed to get a handle on the state explosion problem such that systems of industrial size can be verified by model checking.
- Fault injection is the intentional activation of faults by hardware or software means to be able to observe the system operation under fault conditions. During a fault-injection experiment the target system is exposed to two types of inputs: the injected faults and the input data.
- The sensors and actuators, placed at the interface between the physical world and cyberspace, are physical devices that will eventually fail, just like any other physical device. Therefore it is state-of-the-art that any industrial-strength embedded system must have the capability to detect or mask the failure of any one of its sensors and actuators.

Bibliographic Notes

In the survey article *Software Testing Research: Achievements, Challenges, Dreams* Bertoloni [Ber07] gives an excellent overview of the state-of-the-art in software testing and some of the open research challenges. The research report "Formal Methods and the Certification of Critical Systems" [Rus93] by John Rushby is a seminal work on the role of formal methods in the certification of safety-critical systems.

Review Questions and Problems

12.1 What is the difference between validation and verification?
12.2 Describe the different methods for test-data selection.
12.3 What is a test oracle?
12.4 How does a component provider and component user test a component based system?
12.5 Discuss the different steps that must be taken to investigate a real-world phenomenon by a formal method. Which one of these steps can be formalized, which cannot?
12.6 In Sect. 12.4.2, three different levels of formal methods have been introduced. Explain each one of these levels and discuss the costs and benefits of applying formal methods at each one of these levels.
12.7 What is model checking?
12.8 What is the "probe effect"?
12.9 How can the "testability" of a design be improved?
12.10 What is the role of testing during the certification of a ultra-dependable system?
12.11 Which are the purposes of fault-injection experiments?
12.12 Compare the characteristics of hardware and software fault-injection methods.