Chapter 12
Validation

Overview ... 1
12.1 Building a Convincing Safety Case ... 3
12.2 Formal Methods ... 7
12.3 Testing ... 11
12.4 Fault Injection ... 15
12.5 Dependability Analysis ... 20

Points to Remember ... 23

Overview

- Is this system fit for its purpose?

- System must be proven to be trustworthy
- Convincing safety case
- Formal methods for ultra-dependable real-time system
- Testing real-time systems, test data selection
- Dependability analysis, hazards and risks, FTA and FMEA

12.1 Building a Convincing Safety Case

A safety case is a combination of a sound set of arguments concerning the safety of a given design.
The safety case must convince an independent certification authority that the system is safe to deploy.

Outline of the Safety Case

Computer systems can fail for external or internal reasons.
External reasons are related to the operational environment (stress, EMI, temperature) and to the system specification.
Internal reasons are

1. Computer hardware failure due to random physical fault. Remedy as part of the safety case: redundancy.
2. Hardware or software contains residual design faults. This is the real challenge. No single validation technology can provide the required evidence that a computer system will meet the ultra-high dependability requirement (see chapter 1).

The safety case must thus combine evidence from independent sources to convince the certification authorities that the system is safe to deploy. These sources are:

1. A disciplined software development process with inspections and design reviews
2. Experimental evidence from testing combined with structural arguments about the partitioning of the system in autonomous error-containment regions.
3. Formal analysis of critical properties

4. Experienced dependability of previous generations of similar systems.
5. Experimental data about field-failure rates of critical components as input for reliability models that demonstrate that the system will mask random component failures with the required high probability.
6. Hardware and design diversity to reduce probability of common-mode design failures

Properties of the Architecture

No single fault must cause a catastrophic failure. For fail-safe applications, safe state must be entered before the consequences of error affect system behavior. For fail-operational applications, safe system service must be provided even after a single fault.

Error Containment Regions: At architectural level, it must be demonstrated that every single fault can only affect a defined error containment region, and will be detected at the boundaries of this error containment region.

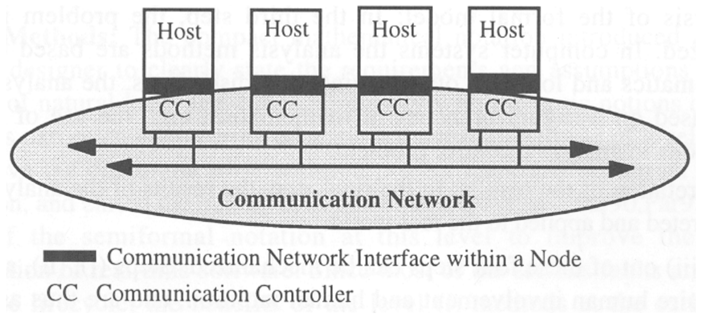
Typical weak points that can lead to common-mode failure of all nodes within a distributed system:

- A single source of time, such as a central clock.
- A babbling node.
- A single fault in the power supply or the grounding system.
- A single design error that is replicated when the same hardware or system software is used in all nodes.

12.1 Building a Convincing Safety Case

ask with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

Example: Four nodes connected by a replicated bus. If communication controller implements an event-triggered protocol, a single faulty host can disrupt communication between all nodes by sending high-priority messages at arbitrary points in time.
If the ARINC 629 protocol is implemented, errors can still occur if the communication controller accesses the bus randomly. In the TTP a bus guardian would prevent such an error.



Host
Host
Host
Host
CC
CC
CC
CC
Communication Network
▬ Communication Network Interface within a Node
CC Communication Controller

12.1 Building a Convincing Safety Case

Composability: Helps in designing a convincing safety case. Assume the nodes can be grouped into nodes implementing safety-critical functions, and into nodes that do not contain safety-critical functions.
If it can be shown, at the architectural level, that no error in any one of nodes not containing safety-critical functions can affect the proper operation of any node containing safety-critical functions, than it is possible to exclude the nodes that do not contain such functions from further considerations during the safety case analysis.

12.2 Formal Methods

12.2 Formal Methods

Formal methods: the use of mathematical and logical techniques to express, investigate, and analyze the specification, design, documentation, and behaviour of computer hardware and software.

Formal Methods in the Real World

Formal investigation of real-world phenomena requires the following steps:

1. Conceptual model building: Informal first step, see chapter 4, leads to reduced natural language representation of the real-world phenomena under investigation. Note: all assumptions, omissions, or misconceptions that are introduced here will remain in the model, and limit the validity of the conclusions derived from the model (assumption coverage).
2. Model formalization: Natural language representation is transformed into a formal specification language with precise syntax and semantics.
3. Analysis of the formal model: Problem is formally analyzed. In computer systems, the analysis methods are based on discrete mathematics and logic. In other disciplines, analysis methods are based on different branches of mathematics, e.g., differential equations to analyse control problems.
4. Interpretation of the results.

Only step 3 can be mechanized. All other steps involve human involvement and human intuition, and are thus fallible.
Ideal and complete verification environment would look like this:

12.2 Formal Methods

- Express specification in formally defined specification language.
- Write implementation in formally defined implementation language.
- Establish mechanically consistency between specification and implementation (e.g., MDA).
- Ensure that all assumptions of the target machine are consistent with the model of computation as defined by the implementation language.
- Ensure that the verification environment itself is correct.

Classification of Formal Methods

Formal methods can be classified into three levels:

1. Use of concepts and notation from discrete mathematics. Ambiguous natural language statements about requirements and specification of a system are replaced by the symbols and conventions from discrete mathematics and logic, e.g. set theory, relations, and functions.
2. Use of formalized specification languages with some mechanical support tools. A formal specification language with a fixed syntax is used, that allows the mechanical analysis of some properties of the problems expressed in the specification language. However, it is not possible to mechanically generate complete proofs.
3. Use of fully formalized specification languages with comprehensive support environments, including mechanized theorem proving or proof checking. 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 language.

Benefits from the Application of Formal Methods

Level (i) Methods:

- Use of level(i) methods improves communication within the project team and within an engineering organization, an enriches quality of documentation.
- Particularly in the early stage of a project the need for precise and effective communication is great, when the mechanical control system and the computer system are specified.
- The concepts of discrete mathematics (set, relation, etc.) are precise and yet abstract. This avoids misunderstandings and ambiguities.
- Some simple mechanical analysis is possible (undefined symbols, uninitialized variables), and can lead to the detection of inconsistencies and omissions.
- Reviews are more effective if requirements are expressed in a precise notation rather than ambiguous natural language.
- It is more difficult to express vague ideas and immature concepts in a semiformal notation.

Level (ii) Methods: a task with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

Level (iii) Methods: a task with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

12.3 Testing

System architecture design is primarily interface design.
An interface between two subsystems of a real-time system is characterized by

- The control properties (control signals crossing interface, what do they do)
- The temporal properties (temporal constraints to be satisfied by control signals and data crossing the interface)
- The functional intent (intended functions of the interface partner)
- The data properties (structure and semantics of data elements crossing the interface)

In case events can occur any time, we speak of a dense time base.
In case events are permitted to occur only in certain time intervals π, we speak of a sparse time base.
In a dense time base it can be difficult to establish temporal order.

Example: The functional intent of a node in an engine controller is to guarantee the car conforms to environmental standards. As the standards change, e.g. new laws are passed, the function of the node may have to change, but not its functional intent.

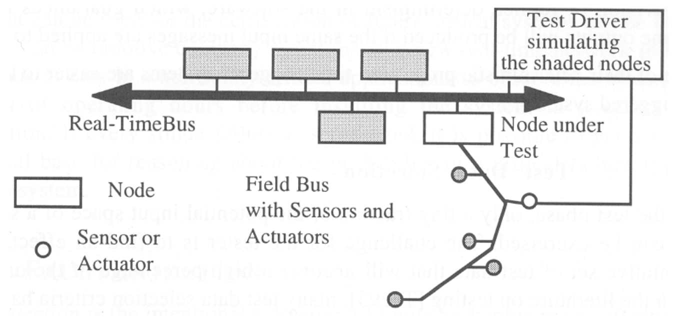
The functional intent is thus at a higher level of abstraction than the function.

The functional intent can be related to a “goal” in requirements engineering.

The Probe Effect

In many cases, the interfacing partners use different syntactic structures and incompatible coding schemes to represent information that must cross the interface.

An intelligent interface component must be placed between the interface partners to transform the different representations. This we call a resource controller.


Test Driver simulating the shaded nodes
Real-TimeBus
Node under Test
Node
Field Bus with Sensors and Actuators
Sensor or Actuator

Resource controllers act as gateways between two different subsystems with different representations.

Design for Testability

12.3 Testing

We distinguish between real world interfaces, and message interfaces.
Example: specific man-machine interface (SMMI) would be a concrete world interface, with touchpad, screen, buttons. A generalized man-machine interface (GMMI or abstract message interface) would just consider the messages that cross the interface, and their temporal properties.

Test Data Selection
To improve compatibility between systems designed by different manufacturers, some international standard organizations have attempted to standardize message interfaces. In automotive systems, the interfaces are typically bus interfaces.

Peak Load: a task with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

Worst Case Execution Time (WCET): a task with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

12.3 Testing

Fault-Tolerant Mechanisms: a task with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

What can be Inferred from “Perfect Working”
To improve compatibility between systems designed by different manufacturers, some international standard organizations have attempted to standardize message interfaces. In automotive systems, the interfaces are typically bus interfaces.

12.4 Fault Injection

12.4 Fault Injection

Example rolling mill with three drive controllers and associated controller nodes and pressure sensors:

Why Fault Injection?
To improve compatibility between systems designed by different manufacturers, some international standard organizations have attempted to standardize message interfaces. In automotive systems, the interfaces are typically bus interfaces.

Automotive transport protocols (OSI level 4)

Transport Protocol	Application	European Standards	US Standards
ISO TP	CAN busses	ISO 15765-2	
SAE J1939	CAN busses		SAE J1939/21
TP 1.6 TP 2.0	CAN busses	Manufacturer standard VW/Audi/Seat/Skoda Base is OSEK COM 1.0	

12.4 Fault Injection

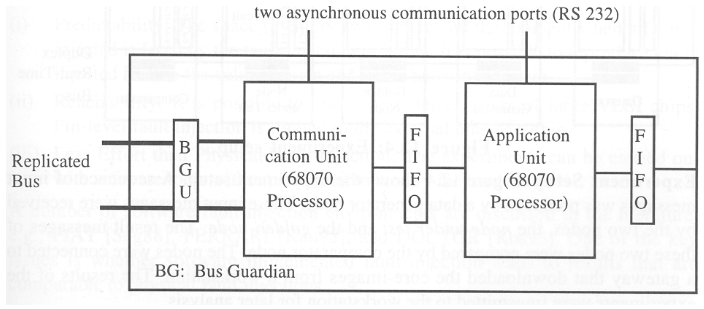
Physical Fault Injection

A system where all control signals are derived from event triggers is called an event-triggered (ET) system. Event examples: button pushed, activation of limit switch, arrival of new message at a node, completion of task within node.

Injected Faults: a task with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

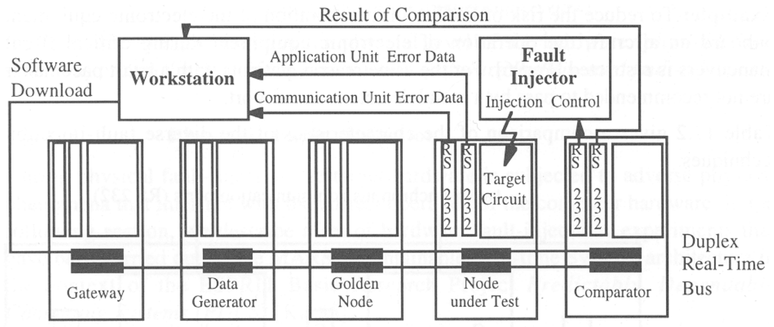
Example: a task with no synchronisation point within, e.g. no semaphore “wait” operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.
A system where all control signals are derived from time triggers is called a time-triggered (TT) system. A time trigger is a control signal derived from the progression of time, e.g. the clock within a node reaches a preset point in time.

Example: computer system controlling an elevator.


two asynchronous communication ports (RS 232)
Replicated Bus
B G U
Communi-cation Unit (68070 Processor)
F I F O
Application Unit (68070 Processor)
F I F O
BG: Bus Guardian

The Hardware under Test: a task with no synchronisation point within, e.g. no semaphore "wait" operation, mutexes or other constructs that could make the task wait or block. Execution time can be determined in isolation, i.e. without knowledge of behaviour of other tasks.

Experimental Setup: a task with no synchronisation point within, e.g. no semaphore "wait" operation, mutexes or



Results: a task with no synchronisation point within, e.g. no semaphore "wait" operation, mutexes or

Software Implemented Fault Injection

In an ET system, the external event triggers are often relayed to the computer system via interrupts. An interrupt is an asynchronous hardware-supported request for a specific task activation caused by an event external (i.e. outside the node) to the currently active computation.

Interrupts cause an overhead (worst case administrative overhead, WCAO), mostly due to the required context switches.

If interrupt frequency is too high, no CPU capacity remains for actual computations. Interrupts are outside the sphere of control of the node, which makes it difficult to handle such overload conditions.

12.5 Dependability Analysis

Deadlines can only be guaranteed if worst case execution (WCET) times of all application tasks are known a priori.

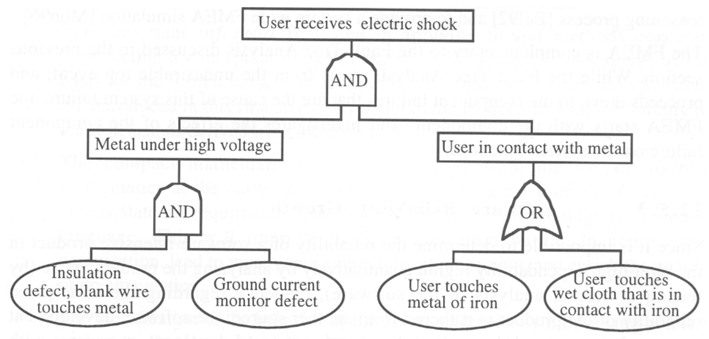
In addition, the worst case delays caused by administrative functions (e.g. operating system services, context switches, scheduling) need to be known. These delays we call the worst case administrative overhead (WCAO).

Fault Tree Analysis

WCET depends on

- source code of task
- properties of object code generated by compiler
- characteristics of compiler

12.5 Dependability Analysis


User receives electric shock
AND
Metal under high voltage
User in contact with metal
AND
OR
Insulation defect, blank wire touches metal
Ground current monitor defect
User touches metal of iron
User touches wet cloth that is in contact with iron

Failure Mode and Effect Analysis (FMEA)

WCET depends on

- source code of task
- properties of object code generated by compiler
- characteristics of compiler

Component	Failure Mode	Failure Effect	Probability	Criticality

Points to Remember

Points to Remember

12.5 Dependability Analysis