TTK4145 Notes

Mikkel Tiller

Part I
Fault tolerance

Four sources of faults in embedded systems:

1. Inadequate specification, i.e. misunderstanding the interactions between the program and environment.
2. Design errors in software components. We typically don't know the consequences of these.
3. Failure of hardware components. More predictable than the one above.
4. Interference in the supporting communication subsystem.

1 Reliability, failure and faults

Reliability is a measure of how well a system conforms to the specification of its behavior. *Response times* are an important part of the specification. **Failure** is when a system deviates from the specification. Highly reliable $\iff$ low failure rate.

The above definitions are concerned with the systems behaviour (external appearance). Failure stems from internal errors, whose algorithmic or mechanical causes are called **faults**. This motivates the following definition of a faulty component: A faulty component is one that under certain circumstances, during the lifetime of the system, results in an error.

An external state that is not in the specification is regarded as a failure, and an internal state that that is not specified is called an error. A fault is **active** when it produces an error, otherwise it is **dormant**. The error propagates through the system, and manifests itself as part of the external behavior.

A failure in one (sub)system can cause faults in other systems, as the following chain of events illustrates:
$\text{Fault} \xrightarrow{\text{activation}} \text{Error} \xrightarrow{\text{propagation}} \text{Failure} \xrightarrow{\text{causation}} \text{Fault}$

With regard to time, there are three kinds of faults:

1. **Transient faults** occur at some point in time, then disappear at some later point in time. E.g. HW response to external electromagnetic field, fault disappears when the field disappears. Common kind of fault in communication systems.

2. **Permanent faults** start at a particular time, then remain in the system until they are fixed. E.g. SW design error.

3. **Intermittent faults** are transient faults that occur from time to time, common with e.g. heat sensitive HW.

Bugs are software faults, and originally there were two kinds:

- **Bohrbugs** are reproducible and identifiable, can be removed during testing or with design diversity techniques.

- **Heisenbugs** are only active under certain rare circumstances, and often disappear when investigated. An example is code shared between concurrent tasks that is not properly synchronized. Heisenbugs can result from software ageing, e.g. not freeing allocated memory, and exhausting the available memory after a long time. Restarting the system clears this bug.

2 Failure modes

A system provides services, and its failure modes can be classified according to their impact on these services. Two general classes:

- **Value failure** - Error in the value associated with service. Can be the result of a data conversion, e.g. 64-bit to 8-bit.

- **Time failure** - Service not delivered at the correct time.

Combinations of the two are called *arbitrary* failures. Time failures can result in the service being delivered:

- too early

- too late (performance error)

- infinitely late (omission failure)

How can a system fail?

1. Fail uncontrolled - arbitrary errors

2. Fail late - delivers service too late in the time domain

3. Fail silent - ommision failure
4. Fail stop - same as above, but permits other systems to detect that it has failed silently
5. Fail controlled - fails in a specified controlled manner
6. Fail never - self explanatory

3 Fault prevention and fault tolerance

Fault prevention aims to eliminate any and all faults before the system goes into operation, whilst **fault tolerance** enables the system to continue functioning even in the presence of faults. Both approaches attempt to give the system well-defined failure modes.

3.1 Fault prevention

Two steps; fault avoidance and fault removal. Fault avoidance consists of:

- Acquiring reliable hardware and protecting it against interference.
- Rigorous specification of requirements, design based on e.g. UML (avsky).
- Use languages with data abstraction and modularity, like Ada and Java.

Fault removal consists of removing causes of errors, mainly by systems tests. However, testing can never remove all potential faults:

- A test can only show the presence of faults, not their absence.
- It might be impossible to test under realistic conditions.
- Errors from the requirement analysis might not become visible until the system is in operation.

Any system will fail eventually (HW or SW), and for real-time systems we need fault tolerance.

3.2 Fault tolerance

Three levels of fault tolerance:

1. **Full fault tolerance** - operation continues without significant loss of performance, but only for a limited time.

2. **Fail soft** - operation continues, but a partial degradation in performance is accepted until recovery or repair.
3. **Fail safe** - the system's integrity is maintained, but a temporary halt in operation is accepted. I.e. the system is shut down in a safe state.

Back in the day fault-tolerant design was based on three assumptions:

1. Algorithms are correctly designed.
2. All failure modes are known.
3. All possible interactions with the environment are known.

This is not realistic today, with multi-core processors and such, hence both anticipated and unanticipated errors must be accounted for.

3.3 Redundancy

Protective redundancy introduces components that detects and recovers the systems from faults, but are unnecessary for normal operation. When designing a fault-tolerant system, the goal is to minimize redundancy while maximizing reliability, subject to constraints on cost, size and power consumption. The redundant components can (and will) increase complexity, and it is useful to separate them from the rest of the system.

We separate between static and dynamic redundancy, both for hardware and software. First, let's have a look at hardware redundancy:

- **Static redundancy** (or masking) is based on redundant components "hiding" faults. An example is Triple Modular Redundancy (TMR), where a majority voting circuit is used. The output of three identical components are compared, and if one differs from the others, its output is masked out. It is assumed that faults are transient.
- **Dynamic redundancy** is an error-detection facility within a component, making it possible for that component to indicate if its output is in error. Note that the component does not hide or fix the error, that must be done by some other part of the system. Examples are checksums (see parity byte section on Wikipedia for very simple example) and parity bits.

For fault tolerance with regards to software design, we have *N-version programming* which works like masking, and *error detection and recovery.* The latter is dynamic redundancy in the sense that recovery is only brought into action once an error has occured.

4 N-version programming

From one initial specification, N independent programs are created. In operation, they run concurrently, and their outputs are compared by a driver process. The "correct" result is determined by majority of vote, like with masking. Challenges concerning N-version programming are:

- **Initial specification** - It is close to impossible to produce unambiguous specifications.
- A complex part of the specification can potentially induce faults for all the N independent developer teams.
- Budget concerns, N-versions are N times more expensive than one.
- **Granularity** - How often results are compared affects overhead in one direction, and fault tolerance in the other.
- **Inexact voting** - Results may not agree exactly, even when no fault has occured. This is a consequence of *finite-precision* arithmetic and the possibility of multiple correct solutions (a quadratic eq. being a simple example). One solution is to regard values inside a range of Θ to be equal, but then the problem arises once again for values close to the boundaries of the range.

5 Software dynamic redundancy

Statically redundant components operate whether or not an error has occured. With dynamic redundancy, however, the redundant components are only put into play when an error occurs. There are four phases to dynamic redundancy in software:

1. **Error detection**
2. **Error diagnosis** - There is a delay between a fault becoming active and error detection, the propagation of erroneous information in the system is assessed.
3. **Error recovery** - Transform the corrupted system into a state where it can continue operation.
4. **Fault treatment** - Maintenance must be performed to correct the underlying fault responsible for the error.

5.1 Error detection

There are two classes:

- **Environmental detection** - Detection by hardware (e.g. overflow error) or run-time support system (e.g. out of bounds error for array).
- **Application detection**
  - **Replication checks** - Check if results are equal.
  - **Timing checks** - Can be a *wtachdog timer* that has to be reset with a given frequency, or detection of missed deadlines by the scheduling system.
  - **Reversal checks** - Compute the input from the output, and compare with the actual input.
  - **Coding checks** - E.g. checksum
  - **Reasonableness checks** - `assert()`-function
  - **Structural checks** - E.g. count number of elements in list to confirm integrity.
  - **Dynamic reasonableness checks** - Error assumed if new output is too different from previous value.

5.2 Error diagnosis

Software designers aim to minimize the damage caused by a faulty component, this is called *firewalling.* Two techniques are:

- **Modular decomposition** - Modules only communicate through well-defined interfaces, internal details are hidden. Provides a static structure.
- **Atomic actions** are indivisible, and appear to happen instantaneously for the rest of the system. Often called **transactions** or atomic transactions. They are used to move the system from one consistent state to another, and limit the flow of information between components/modules.
- There are also **protection mechanisms** which may stop a process from accessing a resource based on its access permissions.

5.3 Error recovery

There are two approaches; **forward** and **backward** error recovery.

Forward error recovery tries to continue from an erroneous state by finding a new consistent (but probably sub-optimal) state. An example is Hamming codes (haven't read about them). Useful if the error related to the

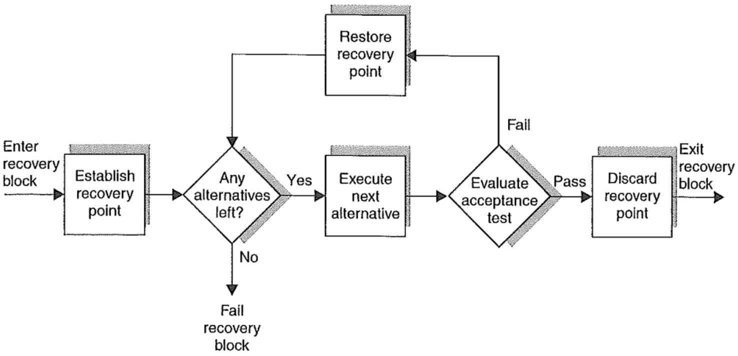

Figure 1: Structure of a recovery block.

previous state may happen many times in a row, and one cannot afford to return to that state.

Backward error recovery restores the system to a previous safe state, a **recovery point**, and executes a different code block than the one that lead to an error. The new code should have the same functionalty, but use a different algorithm. Setting up a recovery point is called **checkpointing**.

Backward recovery can be used to recover from unanticipated faults (very good), but cannot undo effects the fault had on the environment (e.g. launching a missile). Furthermore, it may be costly in a real-time sense to save state from time-varying sensor data.

State restoration with concurrent processes is not necessarily simple, as is seen from the **domino effect**. Let us say we have two processes, say P_1 and P_2 that communicate, synchronize and set up recovery points. If P_1 detects an error at time T_e, it will roll back to its previous safe state. But what if there was communication between P_1 and P_2 between that state and T_e? Then P_2 must also roll back to its previous state, and there might have been some communication in between there as well. This might continue until both processes are back to square one, and is called the *domino effect.*

The probability of the domino effect increases with the number of concurrent processes. A **recovery line**, a consisent set of recovery points, is required to avoid the effect.

6 Recovery blocks

Figure 1 illustrates a recovery block.

6.1 Acceptance test

The acceptance test provides an error detection mechanism, e.g. with invokations of `assert()`. There is always a trade-off between a comprehensive test and affecting the 'happy path' as little as possible. All the error detection techniques discussed in Section 5.1 can be used to create acceptance tests.

7 Comparison between N-version programming and recovery blocks

Brief comparison:

- N-version is static, all versions run regardless of whether an error has occured or not. Recovery blocks are dynamic.
- N-version requires a driver process, while recovery blocks need an acceptance test. At run-time N-version requires N times the resources, since recovery blocks only run one code block at a time. However, establishing recovery points, and reverting to them is expensive.
- Both are prone to errors stemming from ambiguous specifications.
- Acceptance tests may be more flexibe than e.g. an inexact voting scheme.
- The backward error recovery of recovery blocks cannot undo effects on the environment (not impossible to design a system that avoids this problem), whereas N-version requires everything to go through the driver before it can affect the outside world.

8 Dynamic redundancy and exceptions

Exception := the occurence of an error. Telling the 'invoker' about the error condition is called **raising** (or signaling or throwing) and exception, and the invoker then **handles** (or catches) the exception.

Exceptions can be regarded as forward error recovery, as the state is not rolled back, but control is handed over to the exception handler (still, backward recovery can be implemented with exceptions). There is an example in B&W that illustrates some of the controversy surrounding exceptions.