Producing reliable behaviour from unreliable components

Before we begin this PBL session, let us check that you have understood the foundational concepts presented in the previous lectures:

- Dependability, Reliability, Availability, …
- Fault, Error , Failure, Bug, Defect …
- Fail Safe/Stop/Silent
- Fault Tolerance
- Redundancy, n-version programming
- Checkpoint, rollback, rollforward

Producing reliable behaviour from unreliable components

Recommended reading

THE INFLUENCE OF SOFTWARE STRUCTURE ON RELIABILITY, D.L. Parnas, 1975.

System Structure for Software Fault Tolerance, BRIAN RANDELL, 1975.

SOFTWARE RELIABILITY: THE ROLE OF PROGRAMMED EXCEPTION HANDLING, P.M. Melliar-Smith and B. Randell, 1977.

Fault-Tolerant Software, Herbert Hecht, 1979.

The Byzantine Generals Problem, LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE, 1982.

AN EXPERIMENTAL EVALUATION OF THE ASSUMPTION OF INDEPENDENCE IN MULTI-VERSION PROGRAMMING, John C. Knight and Nancy Leveson, 1986

Basic Concepts and Taxonomy of Dependable and Secure Computing, Algirdas Avizienis, Jean-Claude Laprie, Brian Randell and Carl Landwehr, 2004.

A direct path to dependable software, Daniel Jackson, 2009.

Producing reliable behaviour from unreliable components

Problem: consider a deployed system in which bounded queues of integer values are critical components for providing overall behaviour.

The code is written in Java and the required behaviour is specified in an interface and abstract test class:

abstractions.BoundedQueueSpecification

A FIFO bounded queue of integers with a maximum number of elements that must be at least 1.

Version:
1
Author:
J Paul Gibson

tests.JUnit_BoundedQueueSpecification

Unit tests for checking bounded queue behaviour as specified by `BoundedQueueSpecification`

Author:
J Paul Gibson

Producing reliable behaviour from unreliable components

TO DO: Check that you understand the functional requirements based on the unit tests specified in the abstract Junit test class.

Note that the unit tests also include a (very simplisitic) performance requirement:

void tests.JUnit_BoundedQueueSpecification.test_performance()

@Test

Make a million calls to each of the bounded queue methods and check that the delay is no bigger than a second

```
@Test
public void test_performance(){

    long time1;
    long time2;
    time1 = System.currentTimeMillis();

    for (int i=0; i<1000000; i++){
        boundedQ.push(1);
        boundedQ.head();
        boundedQ.is_empty();
        boundedQ.is_full();
        boundedQ.get_size();
        boundedQ.pop();}

    time2 = System.currentTimeMillis();
    System.out.println(time2-time1);
    Assert.assertTrue(time2-time1<1000);
}
```

Producing reliable behaviour from unreliable components

There are 4 *unreliable* implementations:

models.UnreliableBoundedQueue1

The same behaviour as the `ReliableBoundedQueue` except that there is a random problematic delay on the execution of the `ReliableBoundedQueue.push` method that results in the performance of the queue no longer being acceptable - as defined in the test `JUnit_BoundedQueueSpecification.test_performance()`.

models.UnreliableBoundedQueue2

The same behaviour as the `ReliableBoundedQueue` except that there is a random problematic delay on the execution of the `ReliableBoundedQueue.head` method that results in the performance of the queue no longer being acceptable - as defined in the test `JUnit_BoundedQueueSpecification.test_performance()`.

models.UnreliableBoundedQueue3

The same behaviour as the `ReliableBoundedQueue` except that there is a random problematic error on the execution of the `ReliableBoundedQueue.push` method that results in the least significant bit of the integer value received being flipped. The unit test code defined in `JUnit_BoundedQueueSpecification` has a probability of finding the error (when it occurs), depending on the frequency of occurence.

models.UnreliableBoundedQueue4

The same behaviour as the `ReliableBoundedQueue` except that when the queue becomes full the service is disabled for a random period of time (between 100 and 500 milliseconds)
When disabled, method calls which can update the state (i.e. `push` and `pop`) result in an `IllegalStateException`

Producing reliable behaviour from unreliable components

Problem:

For each of the unreliable components you are unable to fix the bugs/change the code. Your task is to try and construct reliable bounded queue behaviour using the unreliable components.

- For each type of unreliable bounded queue, design a mechanism for building reliable bounded queue behaviour from the unreliable behaviour.

- Implement your designs in Java and test them.

Questions:

- Each of the unreliable behaviours is parameterised by some probabilistic/timing/frequency values. For what range of values will your reliability fixes continue to work (and with what degree of reliability)?
- What addtional tests, if any, should your solutions pass that they will/may fail?