Functional Safety and the Use of Java in Embedded Systems

Isabella Stilkerich, Bernhard Sechser
Embedded Systems Engineering Kongress
05.12.2012
Functional Safety

- Ariane 5 (July 4th, 1996)
  - Detonation shortly after takeoff because of an error in the control software
  - Root cause: Insufficient tests of a reused “proven in use” software component
Existing Standards

**IEC 61508**
Functional safety of electrical / electronic / programmable electronic safety-related systems

- **EN 62061**
  - ISO 13849
  - Manufacturing

- **IEC 61513**
  - IEC 60880
  - Nuclear

- **IEC 62304**
  - Medical

- **EN 50271**
  - EN 50402
  - Gas Measuring

- **DO 178B**
  - Aviation

- **IEC 61511**
  - Automation

- **EN 50126**
  - EN 50128
  - EN 50129
  - Rail

- **ISO 26262**
  - Automotive

- **IEC 61513**
  - IEC 60880

- **Nuclear**

- **ISO 13849**

- **DO 178B**

- **IEC 61513**

- **IEC 60880**

- **Manufacturing**

- **Gas Measuring**

- **Aviation**

- **Rail**

- **Medical**

- **Automotive**

...
### ISO 26262

#### 1. Vocabulary

- Overall safety management

#### 2. Management of functional safety

- Safety management during the concept phase and the product development
- Safety management after the item's release for production

#### 3. Concept phase

- Item definition
- Initiation of the safety lifecycle
- Hazard analysis and risk assessment
- Functional safety concept

#### 4. Product development at the system level

- Specification of the technical safety requirements
- System design

#### 5. Product development at the hardware level

- Initiation of product development at the hardware level
- Specification of hardware safety requirements
- Hardware design
- Evaluation of the hardware architectural metrics
- Evaluation of the safety goal violations due to random hardware failures
- Hardware integration and testing

#### 6. Product development at the software level

- Software architectural design
- Software unit design and implementation
- Software unit testing
- Software integration and testing
- Verification of software safety requirements

#### 7. Production and operation

- Production
- Operation, service (maintenance and repair), and decommissioning

#### 8. Supporting processes

- Interfaces within distributed development
- Specification and management of safety requirements
- Configuration management
- Change management
- Verification
- Documentation
- Confidence in the use of software tools
- Qualification of software components
- Qualification of hardware components
- Proven in use argument

#### 9. ASIL-oriented and safety-oriented analyses

- Requirements decomposition with respect to ASIL tailoring
- Criteria for coexistence of elements
- Analysis of dependent failures
- Safety analyses

#### 10. Guideline on ISO 26262
Hazard Analysis and Risk Assessment

- Goal: Risk reduction to an acceptable level

- Residual risk
- Tolerable risk
- Necessary risk reduction
- Actual risk reduction
- Partial risk covered by other risk reduction measures #2
- Partial risk covered by E/E/PE safety-related systems
- Partial risk covered by other risk reduction measures #1
- Risk reduction achieved by all the safety-related systems and other risk reduction measures
System Design (ISO 26262: 4-7)

• Systematic Failures
  • are already in the system at commissioning time
  • manifest themselves under certain circumstances
  • can affect the safety of a system directly
  • may have an impact on all relevant components (hardware, software, etc.)

• Random Failures
  • Are not a priori in the system
  • Arise only after a non-quantified, random or apparently random time
  • random errors appear usually only in the operation of the hardware

• Goal: A dependable runtime system for the application of software-based fault tolerance (FT) measures
Goals

- Automatic application of FT measures
- Ensurance of runtime system dependability
Functional Safety

• e.g. IEC 61508, domain-specific standards
• System consists of hardware (HW) and software (SW)
• A system can have systematic and random faults
  • Systematic errors have to be avoid or mitigated (HW and SW)
  • Random errors can only be mitigated (HW only)
    • by means HW measures (ECC etc) or SW measures
• Objective: A dependable runtime system for the application of fault tolerance (FT) measures
Motivation
Motivation

- Transient hardware faults become more likely
  - soft error rate in logic has increased by 9 orders of magnitude
  - soft error rate in SRAM is constantly high
  - soft errors cannot be ignored anymore
Motivation

- Transient hardware faults become more likely
  - soft error rate in logic has increased by 9 orders of magnitude
  - soft error rate in SRAM is constantly high
  - soft errors cannot be ignored anymore

- Hardware-based fault tolerance (FT) techniques
  - expensive: size, weight and power
Motivation

• Transient hardware faults become more likely
  • soft error rate in logic has increased by 9 orders of magnitude
  • soft error rate in SRAM is constantly high
  • soft errors cannot be ignored anymore
• Hardware-based fault tolerance (FT) techniques
  • expensive: size, weight and power
• Software-based fault tolerance (FT) techniques
Automatic Application of FT

- Measures differ in protection and costs
- Measures are inherently application-specific

**SW Fault Tolerance**

- Checksums
- Redundancy
- Control Flow Monitoring
- ... FT Data Structures

Function Library

- Seat Control
- Light System
- Wiper System
- ...
- ABS

Configuration

Variant A

Distributed System A

Code

Variant B

Distributed System B

Code

Measures differ in protection and costs

Measures are inherently application-specific
Automatic application of FT

- Compiler-based approach
  - Separation of functional code and FT
  - Configurability of FT
  - FT measure tailored towards the application
- Automatic application of FT possible by means of
  - Static analysis of a static system
  - Type-safe programming language
- Example for a FT technique: n-modular redundancy
- Used framework: The KESO Multi-JVM
Java and Static Embedded Systems

- comprehensive a priori knowledge
  - code
  - system objects (tasks/threads, locks, events)
  - system object relationships (e.g., which tasks access which locks)

- benefits of Java
  - more robust software (cf., MISRA C)
  - software-based spatial isolation

- problems of Java
  - dynamic code loading
    - fullyFeatured Java runtimes (e.g., J2ME configurations)
  - overhead
    - code is interpreted or JIT compiled (execution time)
    - dynamic linking (footprint)
**KESO**

- JVM tailoring (instead of fixed configurations)
  - static applications, no dynamic class loading
  - no Java reflection
  - ahead-of-time compilation to Ansi C, VM bundled with application
- scheduling/synchronization provided by underlying OS
  - currently AUTOSAR/OSEK OS
  - accustomed programming model remains
- Integration with legacy C applications is possible
- smallest system to date: Robertino
  - Autonomous robot navigating around obstacles
  - Control software running on ATmega8535
  - 8-bit AVR, 8 KiB Flash, 512 B SRAM
The KESO Multi-JVM

- Java-to-c ahead-of-time compiler
- VM tailoring static configuration
Memory Safety

```java
public class Average {

    protected int sum, count;

    public synchronized void addValues(int[] values) {
        for(int i=0; i < values.length; i++) {
            sum += values[i];
        }
        count += values.length;
    }

    public synchronized int average() {
        return (sum / count);
    }
}
```

“Memory safety ensures the validity of memory references by preventing null pointer references, references outside an array’s bounds, or references to deallocated memory.”

Aiken et. al (Microsoft Research)
**Memory Safety**

```java
public class Average {
    protected int sum, count;

    public synchronized void addValues(int[] values) {
        for(int i=0; i < values.length; i++) {
            sum += values[i];
        }
        count += values.length;
    }

    public synchronized int average() {
        return (sum / count);
    }
}
```

“**Memory safety** ensures the **validity of memory references** by preventing **null** pointer references, references outside an array’s bounds, or references to deallocated memory.”

Aiken et. al (Microsoft Research)
public class Average {

    protected int sum, count;

    public synchronized void addValues(int values[]) {
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }
        count += values.length;
    }

    public synchronized int average() {
        return (sum / count);
    }
}

“Memory safety ensures the validity of memory references by preventing null pointer references, references outside an array’s bounds, or references to deallocated memory.”

Aiken et. al (Microsoft Research)
“Memory safety ensures the validity of memory references by preventing null pointer references, references outside an array’s bounds, or references to deallocated memory.”

Aiken et. al (Microsoft Research)
public class Average {

    protected int sum, count;

    public synchronized void addValues( int values[] ) {
        for(int i=0; i < values.length; i++) {
            sum += values[i];
        }
        count += values.length;
    }

    public synchronized int average() {
        return (sum / count);
    }
}

“Memory safety ensures the validity of memory references by preventing null pointer references, references outside an array’s bounds, or references to deallocated memory.”

Aiken et. al (Microsoft Research)
public class Average {

    protected int sum, count;

    public synchronized void addValues( int values[] ) {
        for(int i=0; i < values.length; i++) {
            sum += values[i];
            count += values.length;
        }
    }

    public synchronized int average() {
        return (sum / count);
    }
}

“Memory safety ensures the validity of memory references by preventing null pointer references, references outside an array’s bounds, or references to deallocated memory.”

Aiken et. al (Microsoft Research)
“Type safety ensures that the only operations applied to a value are those defined for instances of its type.”

Aiken et. al (Microsoft Research)
**Type Safety**

```
public class Average {

    protected int sum, count;

    public synchronized void addValues( int values[] ) {
        for(int i=0; i < values.length; i++) {
            sum += values[i];
        }
        count += values.length;
    }

    public synchronized int average() {
        return (sum / count);
    }
}
```

"**Type safety** ensures that the only operations applied to a value are those defined for instances of its type."

*Aiken et. al (Microsoft Research)*
Application measures

- Implementation of FT measures based on
  - Control flow analyses
  - Data flow analyses
  - Rapid type analyses (e.g. allows for high-level method devirtualization)
  - Fault tolerant data structures
- Efficient application of FT measures through JVM tailoring
- Support for the operating system (OS)
  - Analyse application and pass information to OS
    - System calls, native library usage as well as hardware access
    - Application state etc.
  - Existing and tested operating systems can be used
- Approach supports safety kernel concept
Goals

- Automatic application of FT measures
- Ensurance of runtime system dependability
Runtime system dependability

- Type safety of programming language
  - Valid references with correct type information
  - Maintain spatial isolation
- Memory management
- Safe communication
  - Portal service
  - Shared memory
  - Native Interface
- Runtime system data
  - e.g. domain descriptor, dispatch table
Type-safety of the language

- Java’s type-safety ensures correct memory access in the absence of HW faults
  - A program can only access memory it has been given an explicit reference to
  - The type of the reference determines in which way the memory is used
- Bit flips can corrupt the integrity of the type system
  - This does not affect the current application only
  - Moreover: SW-based memory protection (null, array checks) are invalidated
    - The error can spread to other software modules and replicas
    - A memory protection may help (MPU)
  - An MPU trap could trigger the recovery of a statically computed state
Type-safety of the language

• Using an MPU
  • Isolation violation causes a trap - ok
  • Some faults within application structures can be found do to an FT technique - ok
  • A fault in the runtime system not causing a trap is not detected and can falsely assume a sane operation - x

• Low-end microcontrollers do not have an MPU

• Ensurance of type safety can render many other FT techniques more efficient or possible at all (at the granularity of objects)
Valid References

- Standard solution only compare the object content (heap)
  - Reference and type information can be corrupted
- References can be enriched via FT information (e.g. checksum)
- Checksum creation is currently implemented in SW
  - A HW operation such as popcount on the x86 architecture is advantageous for the execution time
- Object alignment can arbitrarily be adapted (just needs recompilation)
Valid References

• How can **memory usage** for the additional FT information be optimized?
  • Alignment can leave bits unused
  • Static application can be placed in a certain memory location. An embedded application usually utilized only small part of the address space
  • The microcontroller architecture determines the valid address regions
  • Example: TC 1796, 1 MB RAM
Valid References

• Integrity check
  • At each object access
  • Before existing checks
    • At method call sites
    • Object field access
  • Adaption of current reference
  • Execution time punishment
  • Alternatively: Checking and adaption at reference loads and stores
SRAMs and DRAMs are most susceptible to transient errors, when data is read or written to memory cells.

Solution: Reference check
- Load value into register
- Write data into memory

Manipulation level: JVM layer
Runtime system dependability

- Type safety of programming language
  - Valid references with correct type information
  - Maintain spatial isolation

- Memory management

- Safe communication
  - Portal service
  - Shared memory
  - Native Interface

- Runtime system data
  - e.g. domain descriptor, dispatch table
Memory Management

- Available strategies for heap allocation
  - RestrictedDomainScope (ImmortalMemory/ScopedMemory in RTSJ)
  - Throughput optimized garbage collection (GC)
  - Latency-aware GC
- Stack allocation
  - Escape analysis
- ROM allocation
  - Constant data (data flow analysis)
Heap Allocation

- ImmortalMemory
  - Heap reference

- Real-time GC

- GCs
  - Data structures
    - Static reference array
    - Object layout groups reference fields
    - Henderson linked stack frames

- Methods
  - Scan-and-mark phase
  - Sweep phase
Stack Allocation

- All objects are conceptually allocated on the heap
  - Unreachable objects are reclaimed by the garbage collector (GC)
  - Collector ensures consistency of the object graph
- Escape analysis: stack allocation
  - reduces GC overhead (e.g. no barriers needed)
  - memory is automatically reclaimed, when method returns
  - reduces the need for complex data structures
  - stack pointer (in register) is the only data structure to be protected
  - RTSJ Scoped memory (explicit use by enter() or RealtimeThread constructor)
ROM Allocation

- Transient error susceptibility of EEPROM and flashes
- A lot of Java objects are constant
  - Are not necessarily marked as final (assist programmer)
  - A whole-program analysis determines (aided by available type information), which data does not change to facilitate ROM allocation (error correction supported)
- The runtime must be adapted (objects cannot be easily moved to read-only memory since the GC has to mark visited objects)
- In combination with stack allocation and ImmortalMemory: may erase the need for a GC
Conclusion

• A multi-JVM approach for safety-critical embedded systems
  • Software-based isolation
  • Combinable with MPU protection
  • Legacy application support
• Type-safe languages and common programming errors
• Java can be as efficient as in C
  • In static embedded systems
  • Static analyses and whole-program optimizations
• Allows for configurable FT measures
  • Application does not to be changed
  • Evaluation of costs vs safety level (Fault injection experiments)
• Tailored runtime can efficiently be hardened against soft errors