KESO
Functional Safety and the Use of Java in Embedded Systems

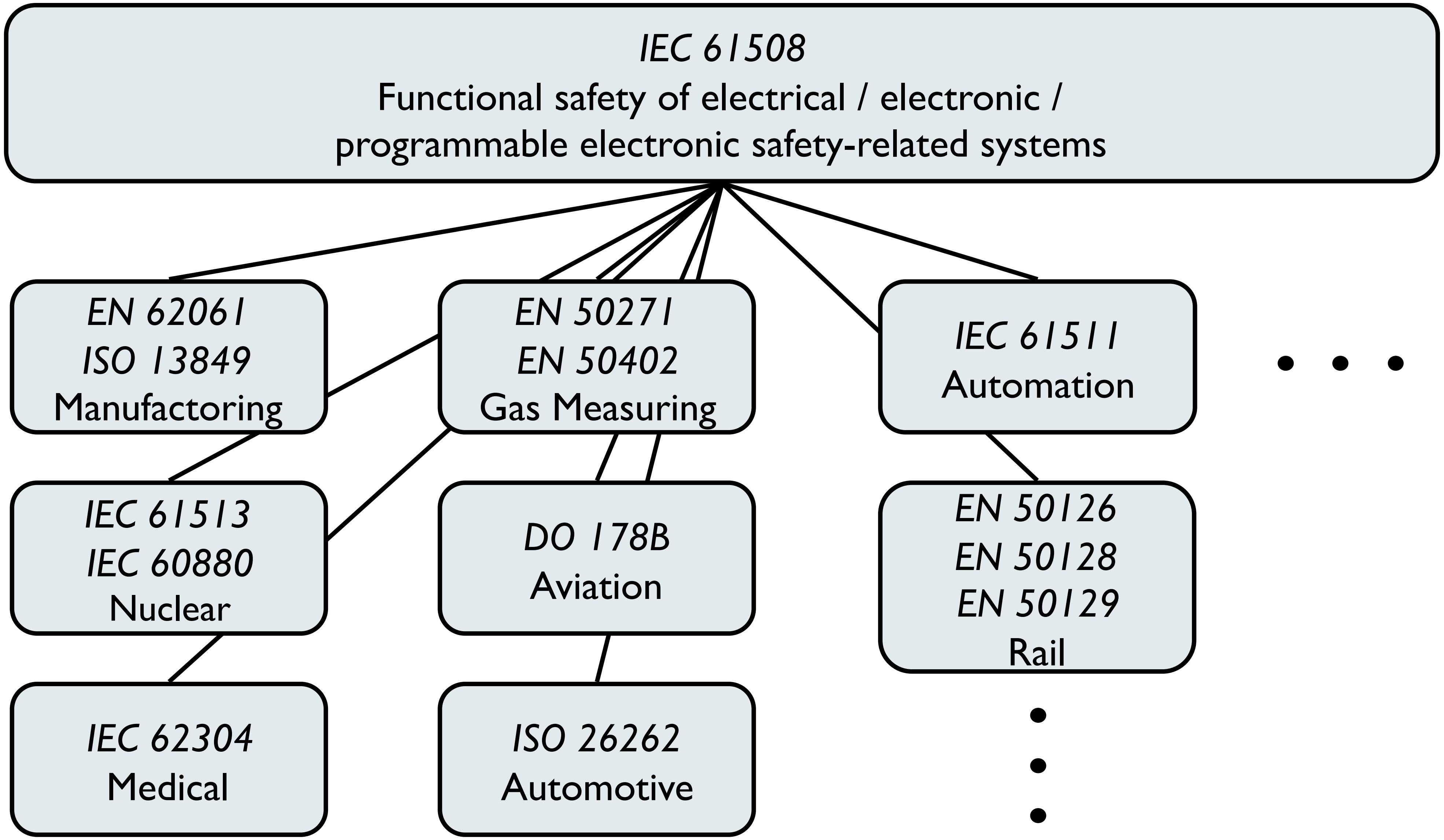
Isabella Stilkerich, Bernhard Sechser
Embedded Systems Engineering Kongress
05.12.2012

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
methodpark
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

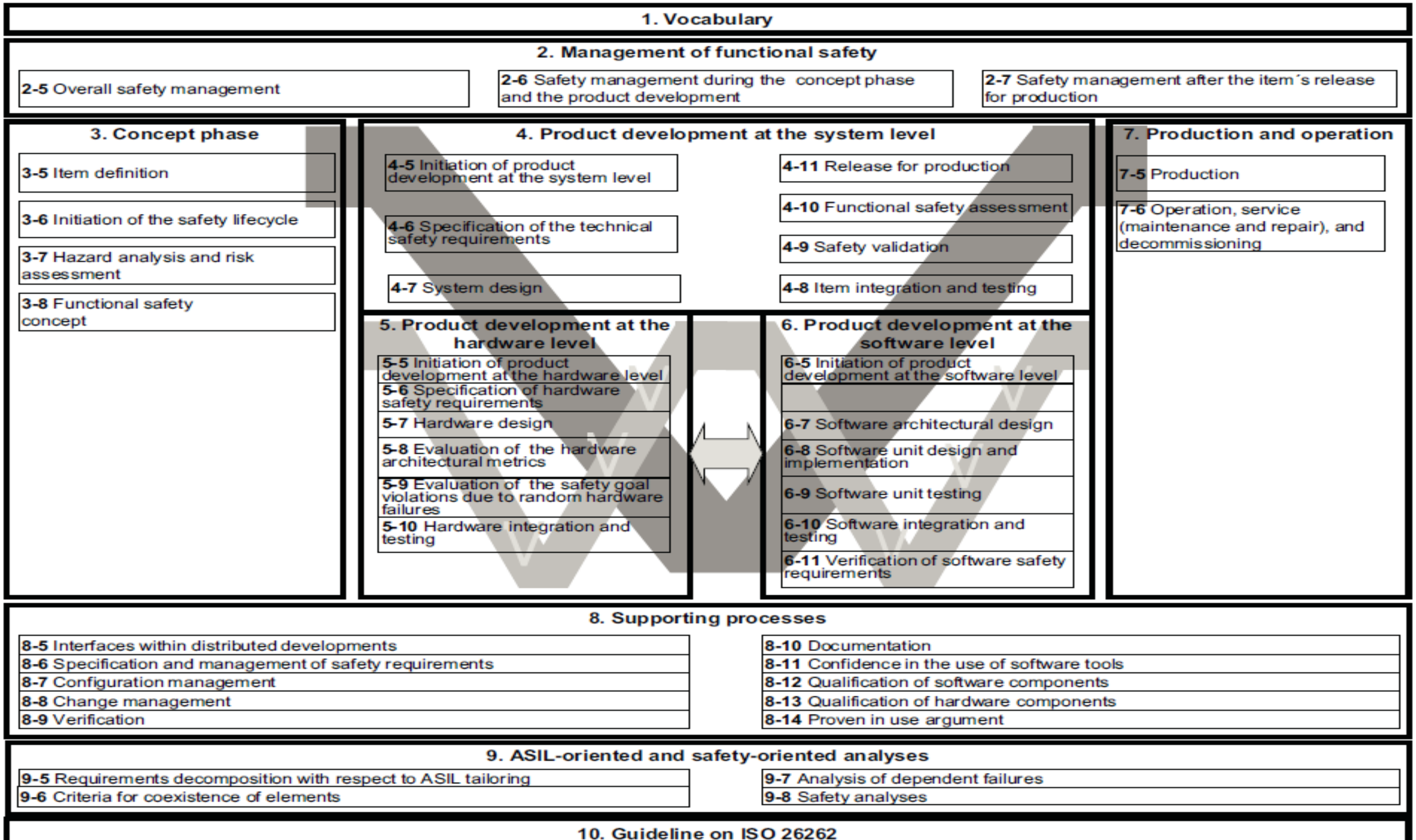
Functional Safety

- 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

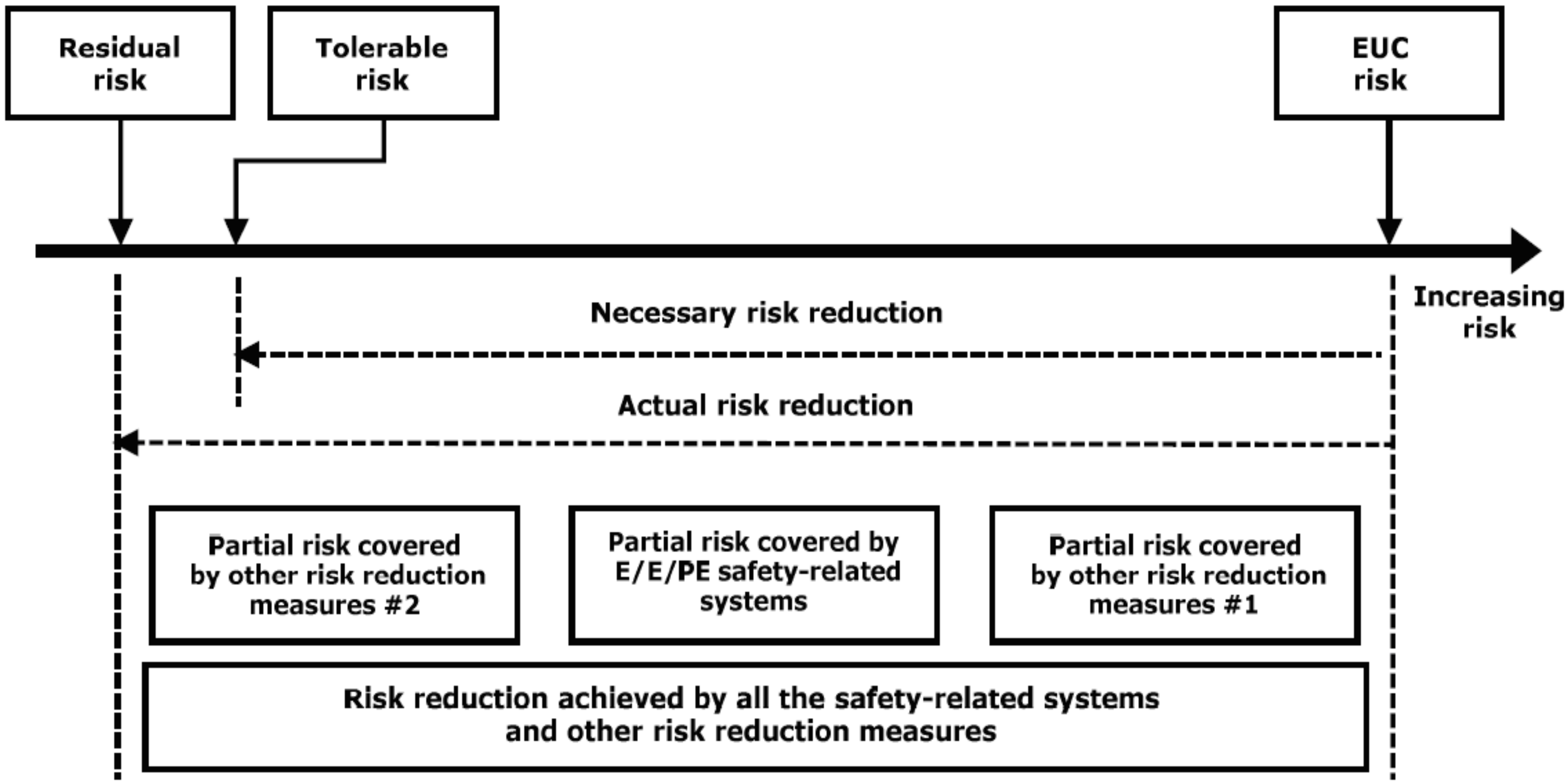


ISO 26262


1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-11 Release for production
4-10 Functional safety assessment
4-9 Safety validation
4-8 Item integration and testing
7. Production and operation
7-5 Production
7-6 Operation, service (maintenance and repair), and decommissioning
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration and testing
6-11 Verification of software safety requirements
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Hazard Analysis and Risk Assessment

- Goal: Risk reduction to an acceptable level



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*

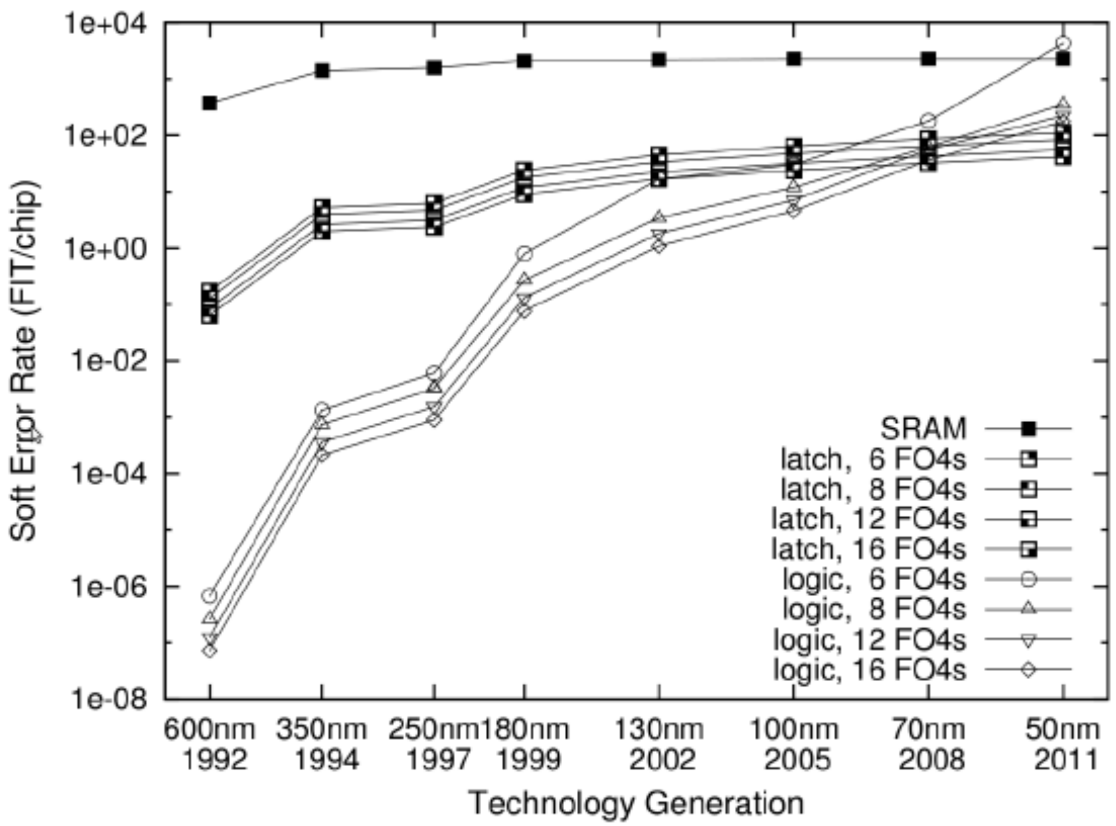
Goals

- 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

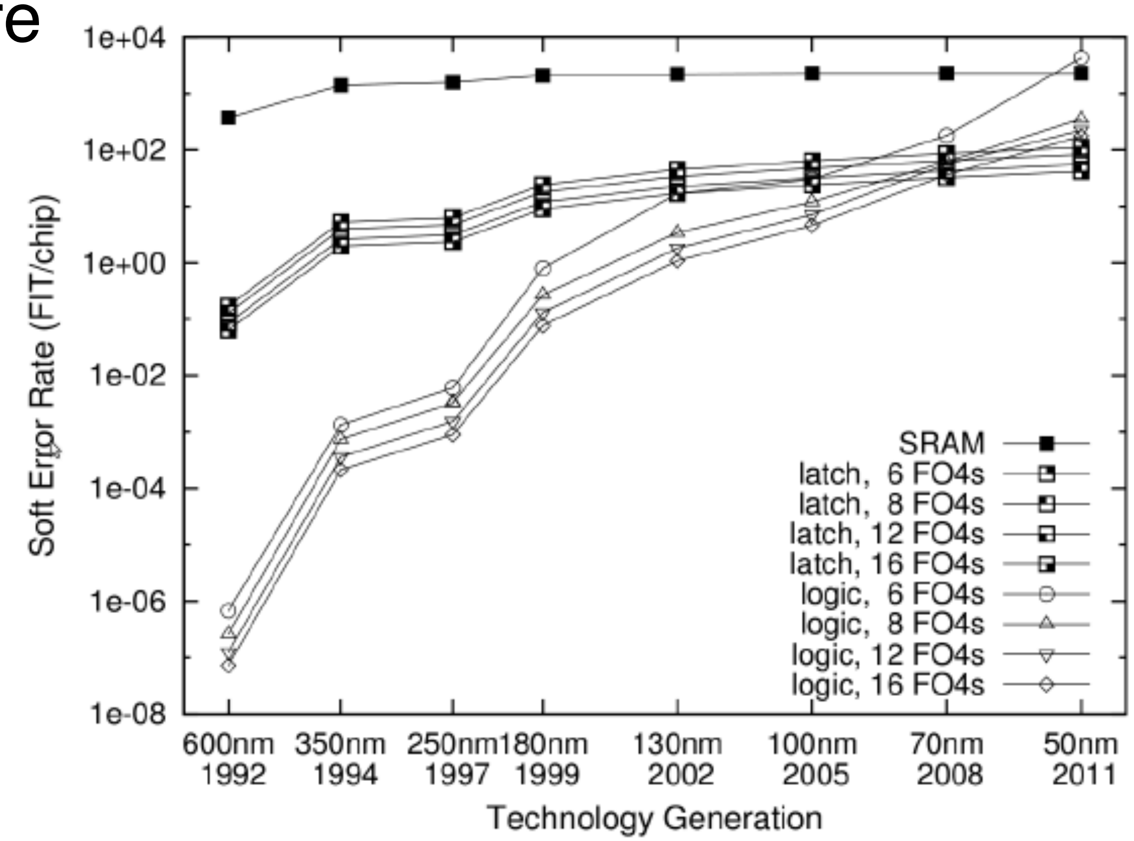
Motivation



1e+04
1e+02
1e+00
1e-02
1e-04
1e-06
1e-08
Soft Error Rate (FIT/chip)
SRAM
latch, 6 FO4s
latch, 8 FO4s
latch, 12 FO4s
latch, 16 FO4s
logic, 6 FO4s
logic, 8 FO4s
logic, 12 FO4s
logic, 16 FO4s
600nm 1992
350nm 1994
250nm 1997
180nm 1999
130nm 2002
100nm 2005
70nm 2008
50nm 2011
Technology Generation

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

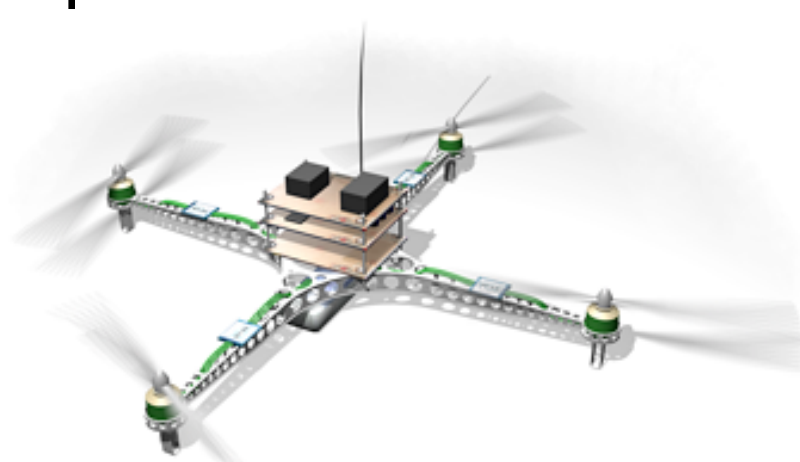


Motivation

- 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

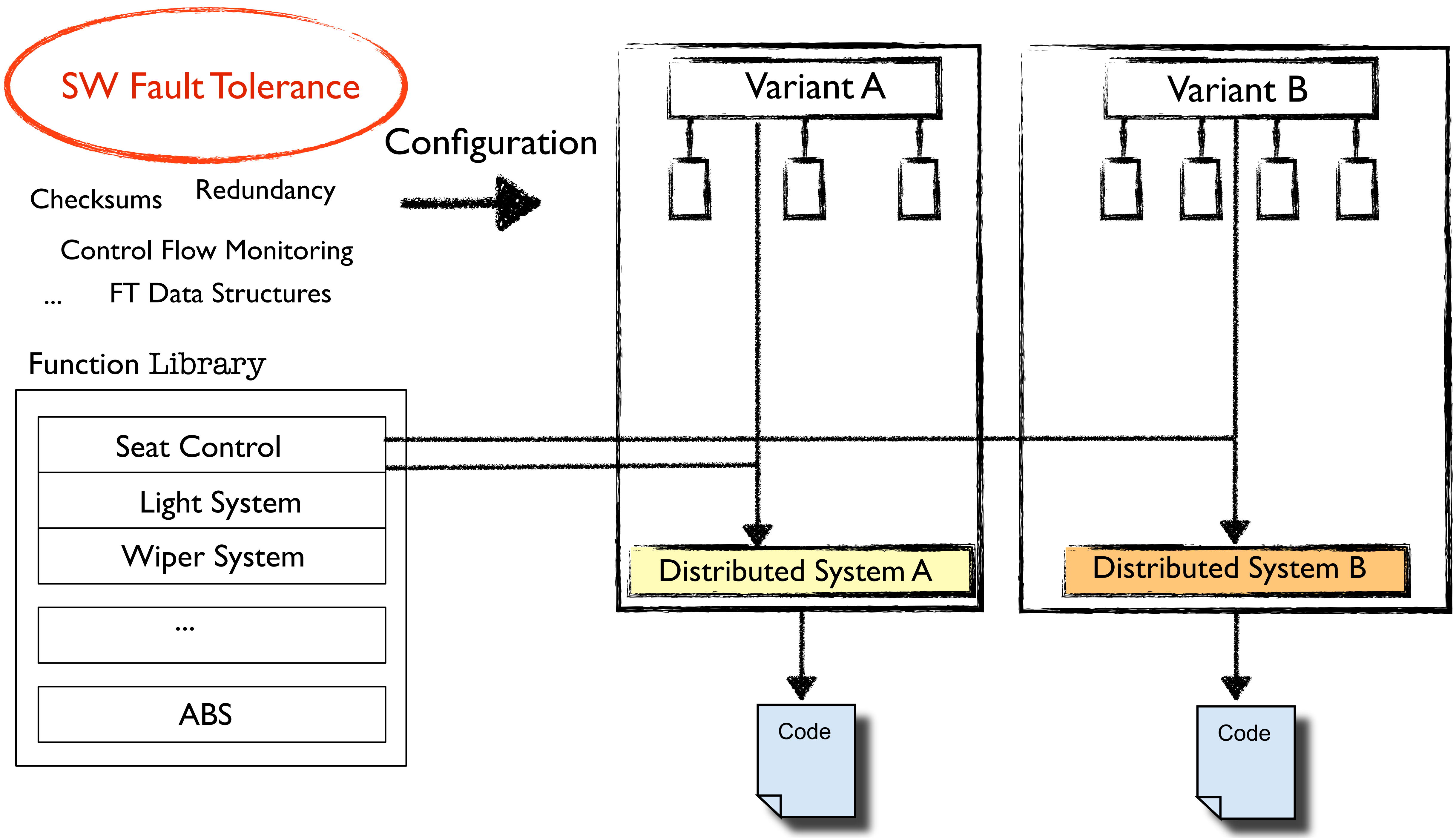
Motivation

- 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

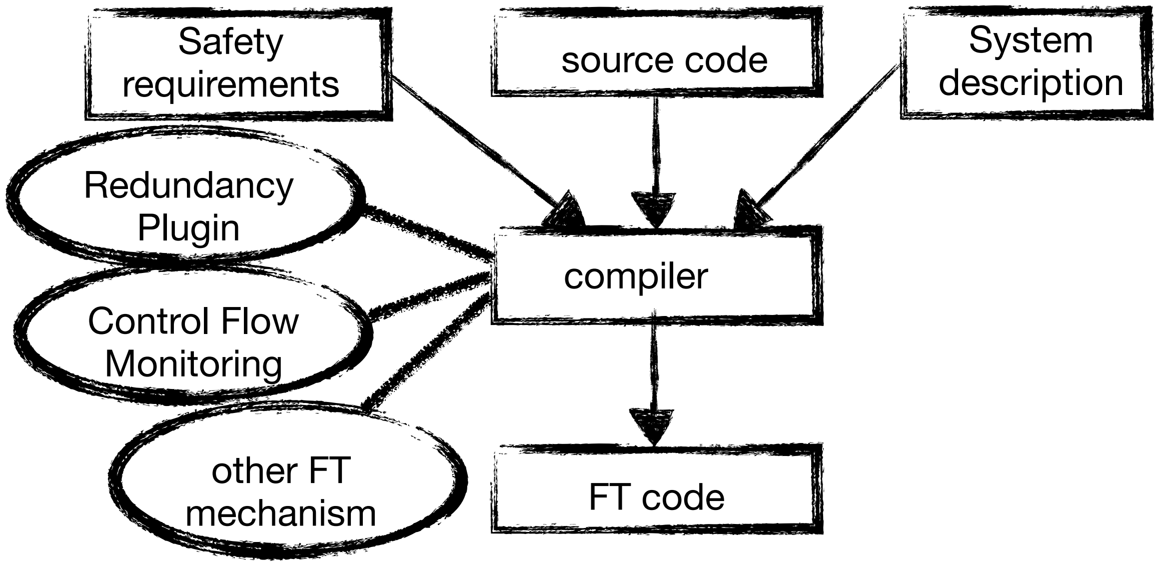


Automatic Application of FT

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



Automatic application of FT



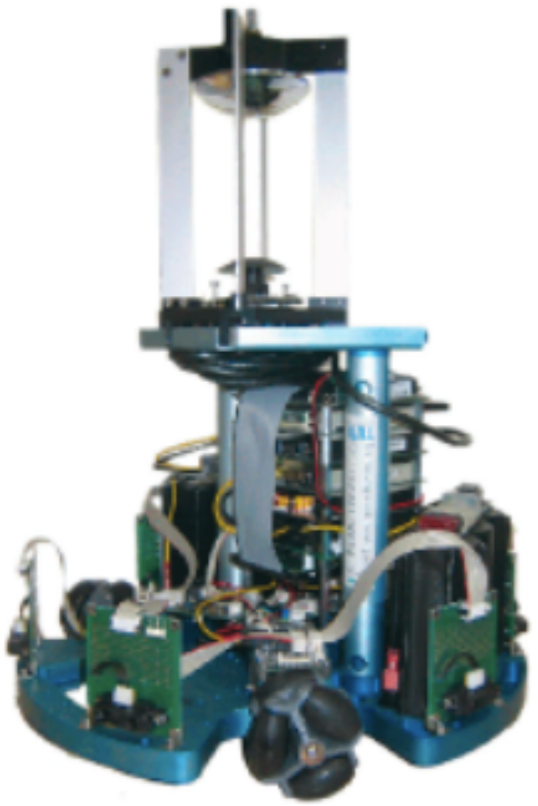
- Compiler-based approach
  - Separation of functional code and FT
  - Configurability of FT
  - FT measure tailored towards the application
- Automatic application of FT possible by means of
  - Static analysis of a static system
  - Type-safe programming language
- Example for a FT technique: n-modular 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
    - fully-featured Java runtimes (e.g., J2ME configurations)
  - overhead
    - code is interpreted or JIT compiled (execution time)
    - dynamic linking (footprint)

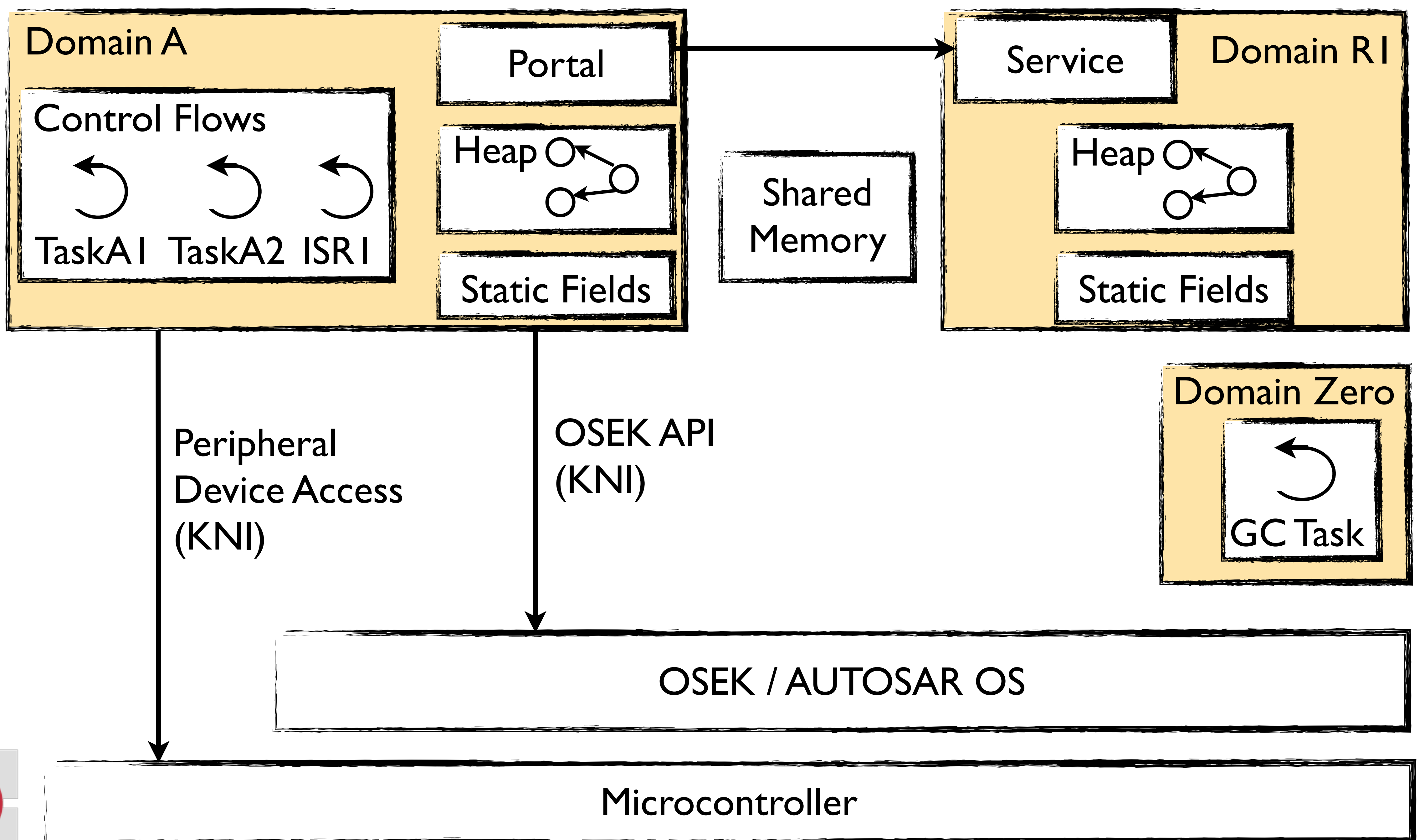
KESO

- 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

```
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

```
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);
  }
}
```

`null != values`

"**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

```
public class Average {

  protected int sum, count;

  pu…ized void …values( int values[] ) {
    … < values.length; i++) {
      sum += values[i];
    }
    count += values.length;
  }

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

null != values

null != values

"**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

```
public class Average {

  protected int sum, count;

  pu…      …ized void …alues( int values[] ) {
    …i < values.length; i++) {
      sum += values[i];
    }
    count += values.length;
  }

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

null != values

null != values

0 <= i < values.length

“**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

```
public class Average {

  protected int sum, count;

  pu...  ...ized void ...values( int values[] ) {
    ...  < values.length; i++) {
      sum += values[i];
    }
    count += values.length;
  }

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

null != values

null != values

0 <= i < values.length

null != values

"**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

```
public class Average {

  protected int sum, count;

  pu[...]ized void [...]values( int values[] ) {
    [...]i < values.length; i++) {
      sum += values[i];
    }
    count += values.length;
  }

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

null != values

null != values

0 <= i < values.length

null != values

count != 0

“**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

“**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;
```

Wert einer Instanz von *Average*

```
  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);
  }
}
```

Operationen auf dem Typ *Average*

“**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 oft 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)

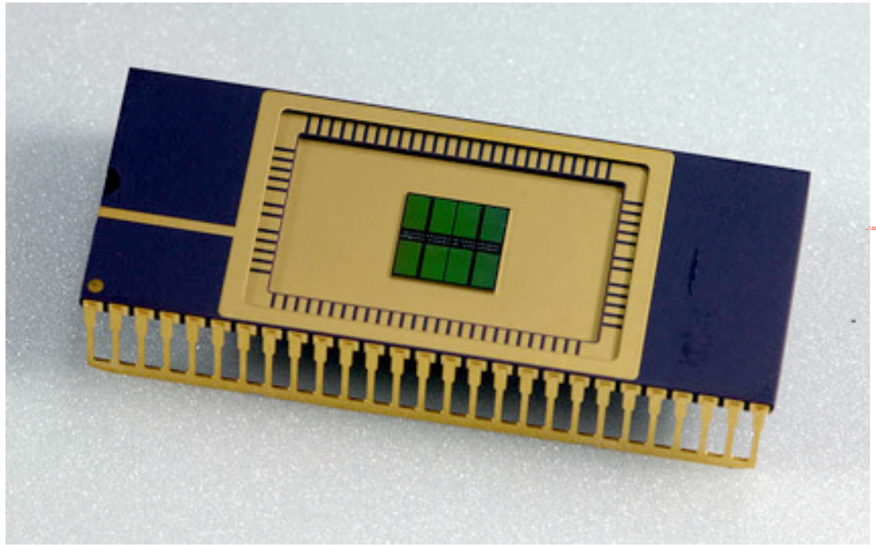
Valid References

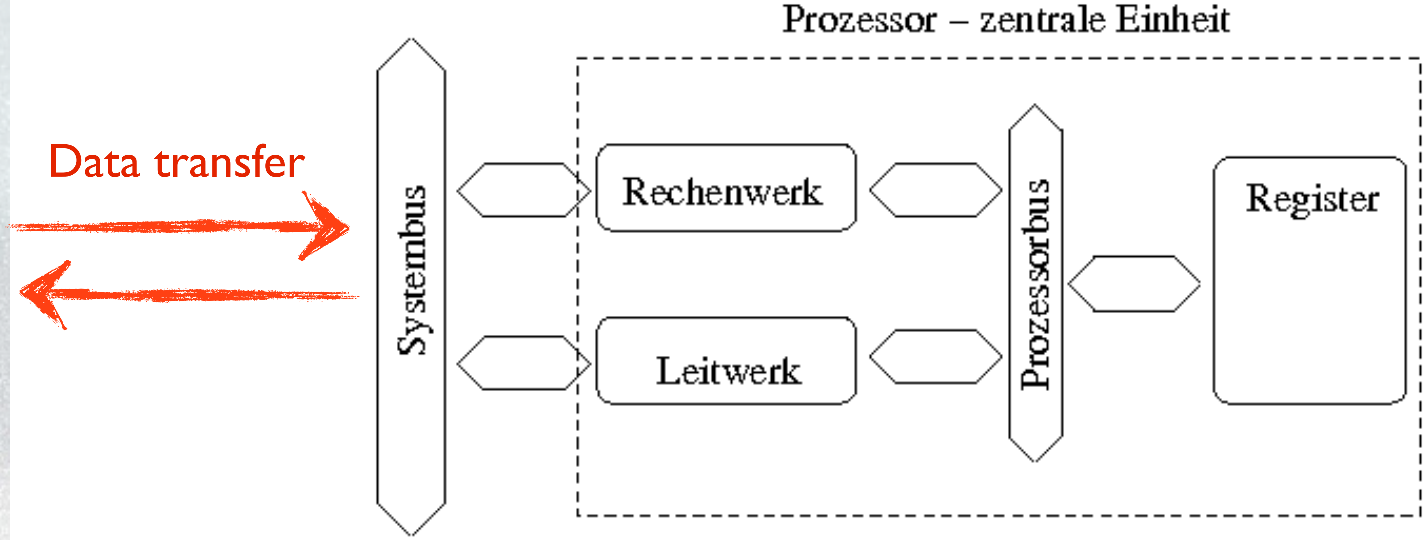
- How can memory usage for the additional FT information be optimized?
  - Alignment can leave bits unused
  - Static application can be placed in a certain memory location. An embedded application usually utilized only small part of the address space
  - The microcontroller architecture determines the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

Valid References

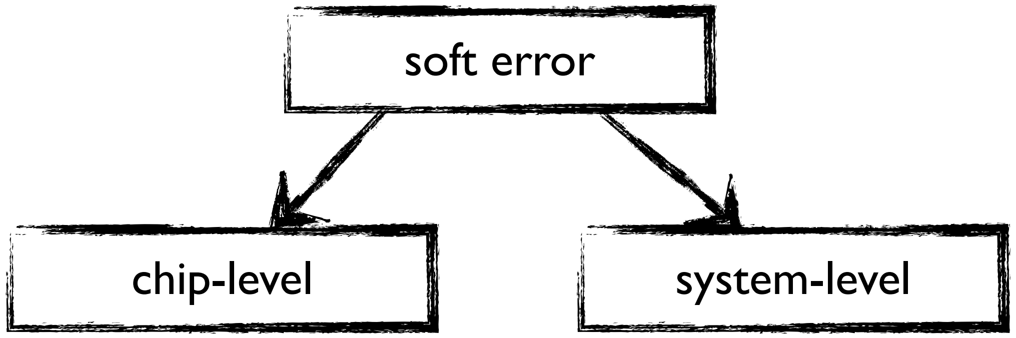
- Integrity check
  - At each object access
  - Before existing checks
    - At method call sites
    - Object field access
  - Adaption of current reference
  - Execution time punishment
  - Alternatively: Checking and adaption at reference loads and stores

Fault susceptability





- SRAMs and DRAMs are most susceptable 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 susceptability 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 faciliate 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