MATLAB EXPO 2016
Rückwirkungsfreiheit zwischen Embedded SW-Komponenten – Polyspace hilft!

By Christian Guß
Freedom of Interference

What is that?

When processes and modules working together on shared resources some interference issues could occur which are very hard to find...

Timing and Execution
- Deadlocks
- Race conditions
- Sequence error

Memory
- Corruption of content
- Access out of bounds
- Invalid r/w access

Exchange of Information
- Interface violation
- Non initialized data
- Null-Pointers
- Data size mismatch
Typical Automotive Software Architecture

- **Hardware**
  - Microcontroller Abstraction Layer
  - ECU Abstraction Layer
  - Services Layer
  - Runtime Environment
  - Application Layer
    - Application 1
    - ... (N applications)

- **Basic Software**
  - External Autosar Interface
  - Services
  - Communication
  - Operating System
    - Internal Interface

- **Non Critical**
  - Affects?

- **Critical**
  - Affects?

- **Failure**

MATLAB EXPO 2016
ISO 26262-6: Freedom from interference (Annex D)

**Goal:** Prevent or detect faults that can cause interference between software elements (e.g. different software partitions)

**D2.2 Timing and execution**
- Deadlocks
- Race Conditions

**D2.3 Memory**
- Corruption of content
  - Out-of-bound pointers and arrays, etc.
- Read or write access to memory allocated to another software element
  - Exhaustive identification of unprotected shared variables
  - Documentation of read-/write access to global variable

**D2.4 Exchange of information**
- Corruption of information
- Loss of information
What you could do is…

Problem: Testing, Hardware protection, restrictions and functional protection could be:
- very expensive to implement,
- not completely protective,
- reducing performance.

(see ISO 26262-6 Annex D)
Let’s make an example...

**Task 1**
Write bad_glob

**Part 1**

**Task 2**
Read bad_glob

**Part 1**

**Part 2**

**Data Race**

**Fix:** Critical Section!

**Problem:** When needed?

Overusing can degrade system performance!

```c
/* DATA RACE
   * Defect Data race */
int bad_glob1; /* Non-atomic write access */
void bug_datarace_task1(void)
{
  bad_glob1 = 1; /* Non-atomic write access */
}
void bug_datarace_task2(void)
{
  int local_var;
  local_var = bad_glob1; /* Non-atomic read access */
  printf("%d", local_var);
}
```
How to reduce efforts with „Timing and Execution“ Safety?

With static analysis!
Polyspace – Data race checks

Find **Timing Issues** with Multitasking

- **Write #1** (non-atomic)
  - Operations may involve multiple machine instructions

- **Read #1** (non-atomic)
  - Operation with 64-bit variable on a 32-bit target

```
long long bad_glb2;
void bug_task3(void)
{
    bad_glb2 += 1;
}
```

```
long long local_var;
local_var = bad_glb2;
```
**Polyspace - Global Variable Usage Protection**

<table>
<thead>
<tr>
<th>Global Variable</th>
<th>Shared</th>
<th>Protected variable</th>
<th>Not shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potentially unprotected variable</td>
<td>3</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>- Variable: PowerLevel tasks1.c _init_globals()</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>- Variable: SHR4 tasks1.c _init_globals()</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Variable: SHR2 tasks1.c _init_globals()</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protected variable</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Variable: SHR5 tasks1.c _init_globals()</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Variable: SHR tasks1.c _init_globals()</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not shared</td>
<td>2</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>Unused variable</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Variable: second_pai...initialisations.c _init_globals()</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Variable: _huge val huge_val.h _init_globals()</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Used non-shared variable</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Shared protected global variable**
  Global variables shared between multiple tasks and protected from concurrent access by the tasks.

- **Shared unprotected global variable**
  Global variables shared between multiple tasks but not protected from concurrent access by the tasks.

- **Non-shared used global variable**
  Global variables used in a single task.

- **Non-shared unused global variable**
  Global variables declared but not used.
Let's make another example...

```c
char myarray[10];
int VeryImportantData;

void myarray_init(char array[], int array_size)
{
    for (int i = 0; i < array_size; i++){
        array[i] = 0;
    }
}

void integration_context()
{
    // ... before ...
    myarray_init(&array[0], 15);
    // ... behind ...
    lets_use_my_important_data(VeryImportantData);
}
```

- Is it safe to use myarray_init Function?

**NO!**

integration_context impacts myarray_init impacts VeryImportantData

hard to find!
Problem with testing: Tests aren’t exhaustive

“Program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra [1])

How to reduce efforts with „Memory“ Safety?

With static analysis!
Polyspace – Proving Memory Safety

With Polyspace …
you can proof the existence and absence of memory access errors like:

---

### Memory safety

- aims to avoid software errors that cause safety and security vulnerabilities
- dealing with random-access memory (RAM) access,
- such as corruption of content and read/write access to memory allocated by another software element.

Computer languages such as C and C++ that support arbitrary pointer arithmetic, casting, and deallocation are typically not memory safe.
Let’s make one last example…

```c
int16 Add1(int16 ul[], uint16 size)
{
    int16 ret = 0;
    uint16 i = 0;
    for (i = 0; i < size; i++)
    {
        ret += ul[i] << i;
    }
    return ret;
}
```
How to reduce efforts with „Exchange of Information“ Safety?

With static analysis!
Example: Optimize design and architecture

Non Robust Module

External code

Potential Runtime Error inside!!!
Example: Optimize design and architecture

Example: Optimize design and architecture

```
int16 Add1(int16 u1[], uint16 size)
{
    int16 ret = 0;
    uint16 i = 0;
    for (i = 0; i < size; i++)
    {
        ret += u1[i] << i;
    }
    return ret;
}
```

Free from Runtime Errors
Summary

➢ Do you have Multicore applications?
➢ Do you have HW/SW protections?
➢ Do you like to reduce testing effort?

ask for our static analysis solutions TODAY