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

- **Application 1**
- **...**
- **Application N**

**Runtime Environment**
- **Services Layer**
  - **ECU Abstraction Layer**
  - **Microcontroller Abstraction Layer**

**Hardware**

**Drivers**

**Basic Software**
- **External Autosar Interface**
- **External Autosar Interface**
- **External Autosar Interface**

**Services**
- **Internal Interface**
- **Internal Interface**

**Communication**
- **Internal Interface**

**Operating System**
- **Internal Interface**

**Failure**
- **Affects ?**

**Non Critical**
- **Critical**

**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...

Robustness-Testing:
- fault injection
- boundary tests

Hardware protection:
- Memory Protection Unit
- Error Correcting Code

Functional protection:
- Cyclic redundancy check
- redundant storage
- defensive code

Restrictions:
- only static memory allocation
- restricted access to memory

Static analysis:
- Data flow analysis
- Control flow analysis
- Formal analysis

Problem: Testing, Hardware protection, restrictions and functional protection could be:
- **very expensive** to implement,
- **not completely protective**, and
- **reducing performance**.
Let’s make an example…

**Task 1**

Write bad.glob

**Part 1**

**Task 2**

Read bad.glob

**Part 1**

Read bad.glob

**Part 2**

**Fix:** Critical Section!

**Problem:** When needed?

Overusing can degrade system performance!
How to reduce efforts with „Timing and Execution“ Safety?

With static analysis!
Polyspace – Data race checks

Find **Timing Issues** with **Multitasking**

ID 2: Data race
Certain operations on variable 'bad_glob2' can interfere with each other and cause unpredictable value. To avoid interference, operations on 'bad_glob2' must be in the same critical section.

- **Write #1** (non-atomic)
  - Operation could involve multiple machine instructions

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

```
long long bad_glob2;

void bug_task3(void)
{
    bad_glob2 += 1;
}

void bug_task4(void)
{
    long long local_var;
    local_var = bad_glob2;
}
```
Polyspace - Global Variable Usage Protection

- **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.

### Notebook Preview

**Protected Variable**

Variable 'tasks1.SHR' is shared among several tasks. All operations on 'tasks1.SHR' are protected by critical section. Read by task: *tregulate*, Written by task: *server1, server2*

<table>
<thead>
<tr>
<th>Event</th>
<th>File</th>
<th>Scope</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written value: 0</td>
<td>tasks1.c</td>
<td>_init_globals()</td>
<td>30</td>
</tr>
<tr>
<td>Written value: 22</td>
<td>tasks1.c</td>
<td>Tserver()</td>
<td>81</td>
</tr>
<tr>
<td>Read value: 0 or 22</td>
<td>tasks1.c</td>
<td>intregulate()</td>
<td>53</td>
</tr>
</tbody>
</table>
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` which 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:

Out of bounds array index
- Warning: array index may be outside bounds: [0..9]
  - array size: 10
  - array index value: [1..10]

```c
int buffer[10], i = {0};
while (i++ < 10){
  buffer[i] = 0;
}
```

Illegally dereferenced pointer
- Error: pointer is outside its bounds
  - This check may be an issue related to unbounded input values
  - Dereference of parameter ‘p’ (pointer to int 32, size: 32 bits):
    - Pointer is null.

```c
void foo(int *p)
{
  if (p == 0){
    *p = 42;
  }
}
```

Non-initialized local variable
- Error: local variable is not initialized (type: int 32)
  - This check may be a path-related issue, which is not dependent on input values

```c
void foo(int *p)
{
  int va;
  if (p != 0){
    *p = va;
  }
}
```

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

1. **int16** (5) to **int16** (5) through **Saturation1** with Range [0..100]
2. **uint16** from **size**

**Non Robust Module**

- **int16** input to **Abs**
- **uint16** from **Divide**

**Additional Range-Limiting Code**

```c
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**

**Runtime Error**

- **Illegal Dereferenced Pointer**
  - Pointer is within its bounds
  - Dereference of expression (pointer to int 16, size: 16 bits):
    - Pointer is not null.
    - Points to 2 bytes at offset 0 or 2 or 4 or 6 or 8 in buffer of 10 bytes, so is within bounds (if memory is allocated).
    - Pointer may point to variable or field of variable:
      - 'rb_Saturation1', local to function PolySpace_demo_ec61508_step.'
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