

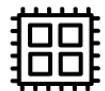


MICROSAR Run-Time Interface

with Lauterbach TRACE32 and TA.Inspection

Introduction

- ▶ The MICROSAR Run-Time Interface with Lauterbach TRACE32 and TA.Inspection for MICROSAR Classic Platform AUTOSAR BSW is intended to provide an efficient way of exporting OS and RTE scheduling traces on hardware traceable microcontrollers aiming to:
 - ▶ Overcome microcontroller specific limitation on traceable amount of cores
 - ▶ Minimize runtime impact on the AUTOSAR project
 - ▶ Minimize impact on the hardware trace interface
 - ▶ Ease hardware tracing setup
 - ▶ Maximum guaranteed freedom of interference to the application code
- ▶ **Supported Hardware Architectures:**
 - ▶ Infineon AURIX TriCore 1st and 2nd Generation
 - ▶ PowerPC



Tracing Workflow

1.

Tracing Basic Knowledge

2.

Configure MICROSAR BSW – OS and RTE

3.

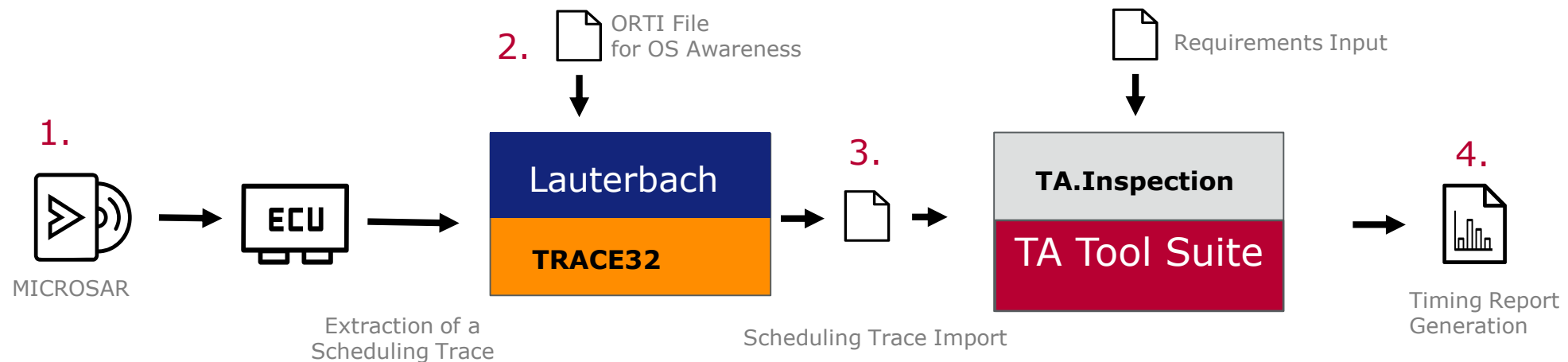
Configure Lauterbach Trace Debugger – TRACE32

4.

TA.Inspection - Trace Evaluation

Example with Lauterbach

1. Configure MICROSAR BSW - Tracing hooks OS and RTE
2. Configure Lauterbach TRACE32 for trace recording and export – ORTI file for OS awareness
3. Import trace into TA Tool Suite
4. Evaluate the traces against requirements



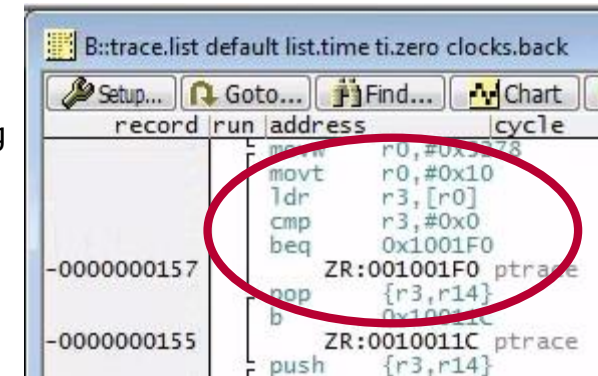
⚙️ Completely automatable for regression tests and troubleshooting

Tracing Basic Knowledge

- ▶ Most evaluation ECU devices include a trace unit, which can be programmed to export trace messages for the program flow and the read/write operations performed. For OS and RTE tracing only data tracing really suits.

1. Program Flow Trace (Instruction Trace)

- ✓ High resolution: assembly instructions can be seen with a timestamp for debugging
- ✗ High bandwidth interface is needed or short trace duration
- ✗ Is not exporting OS information such as Task States



2. Data Trace

- ✓ Read and write accesses to variables can be exported with timestamp
- ✓ Trace Data can be limited to the minimum
- ✗ Os State changes variables need to be available to data trace scheduling behavior

3. Instrumentation Trace

- 🔍 Adding OS state change variables to the project to only trace scheduling behavior
- 🔍 Trace bandwidth and OS overhead is limited to most possible compromise
- 🔍 Trace information is "lifted" from assembly to Software Architecture level (OS and RTE awareness)

Tracing Instrumentation

- ▶ To signalize task state changes and runnable execution we store the information in a dedicated variable for each core
- ▶ Data tracing exports the variable write accesses

OS Instrumentation Example:

- ▶ Those Variables store all relevant Information:
 - ▶ Core ID
 - ▶ Task ID
 - ▶ Task State

```
#define OS_VTH_ACTIVATION(TaskId, DestCoreId, CallerCoreId) \  
{arti_os_trace = (TaskId<<16) | (ARTI_VALID_OS_SIGNALING<<8) | (ARTI_OS_ACTIVATE<<8) | DestCoreId ; /* e.g. ready */ }
```

- ▶ When a task is activated this OS_VTH_ACTIVATION hook is called by the OS
- ▶ In those scheduling hooks we write to the instrumentation variable
- ▶ The ECU recognizes the data access to the instrumentation and exports the trace message to the debugger.

Tracing Workflow

1.

Tracing Basic Knowledge

2.

Configure MICROSAR BSW – OS and RTE

3.

Configure Lauterbach Trace Debugger – TRACE32

4.

TA.Inspection - Trace Evaluation

Installation

1. Copy the CreateLauterbachRunnableMeasurementPoints.dvgroovy into the following SIP folder:
/DaVinciConfigurator/AutomationScripts

2. Copy the gen_vector_rte_hook.py into the DaVinci Project folder.

 gen_vector_rte_hook.py

 VTT_Brake.dpa

3. Adopt the file paths to Rte_Hook.h and to the generated files Rte_Hook_t32.c and rte_runnable.cmm

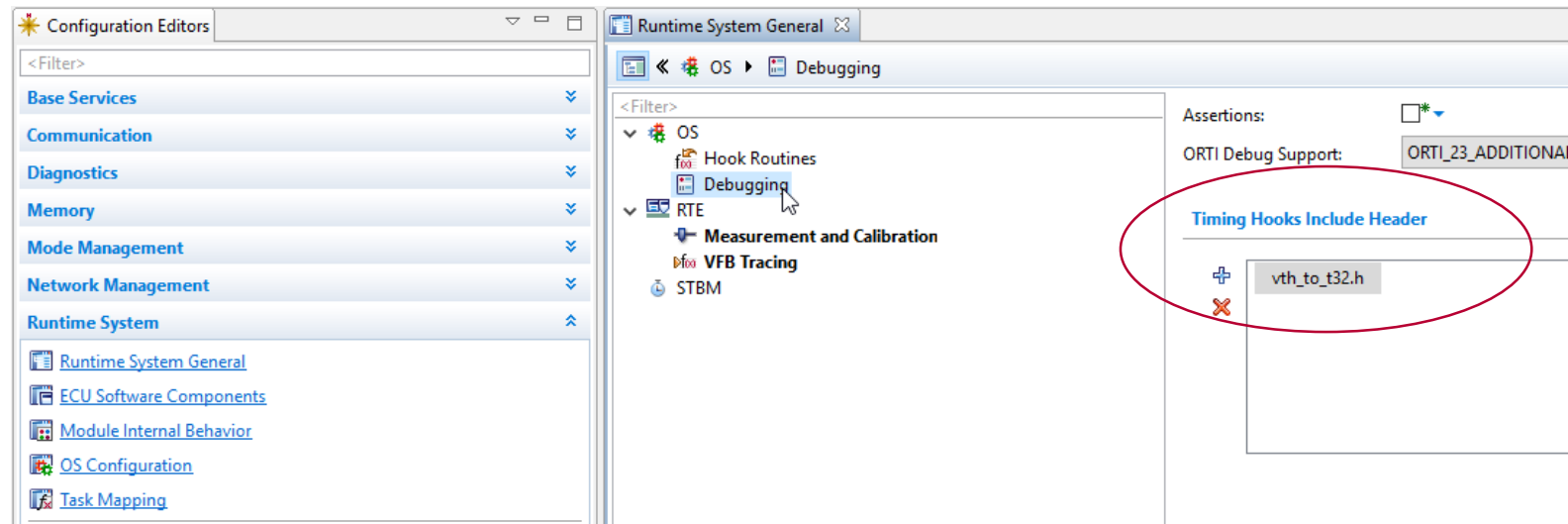
```
fsrc=open("./ApplHw/GenData/Rte_Hook.h",'r')           # Generated by RTE
fdest=open("./Appl/Source/Rte_Hook_t32.c",'w')         # Generated by this Script
cmmdest=open("./Appl/Source/rte_runnable.cmm",'w')     # Generated by this Script
```

4. Add the vth_to_t32.c to your make environment
5. Depending on the target ECU and compiler, the base address for the trace variables needs to be adjusted as well. For HighTec compiler, a linker script example is provided. For Tasking, the compiler pragma __at() is used.
6. At last, activate the OLDA Memory to be used with the following flag in your Init Code:

```
#include <IfxDom_reg.h>
static void __noinline__ __noreturn__ __jump__ _start( void )
{
    // activate OLDA memory
    DOM0_BRCON.B.OLDAEN = 1;
}
```


Configure MICROSAR OS Tracing Hooks

- ▶ OS Vector Timing Hooks are used to signalize Task and ISR State switches
 - ▶ Therefore, a Timing Hooks header needs to be included

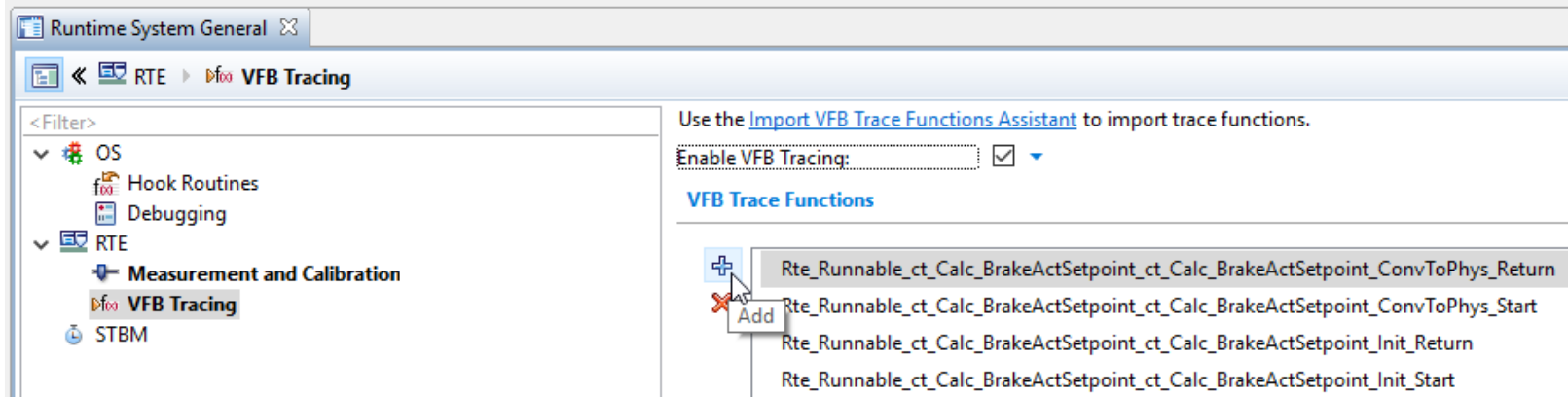


- ▶ The header includes the trace instrumentation is provided by the Trace Vendor (Lauterbach)

```
#define OS_VTH_ACTIVATION(TaskId, DestCoreId, CallerCoreId) \  
{arti_os_trace = (TaskId<<16) | (ARTI_VALID_OS_SIGNALING<<8) | (ARTI_OS_ACTIVATE<<8) | DestCoreId ; /* e.g. ready */ }
```

Configure MICROSAR VFB Tracing Hooks

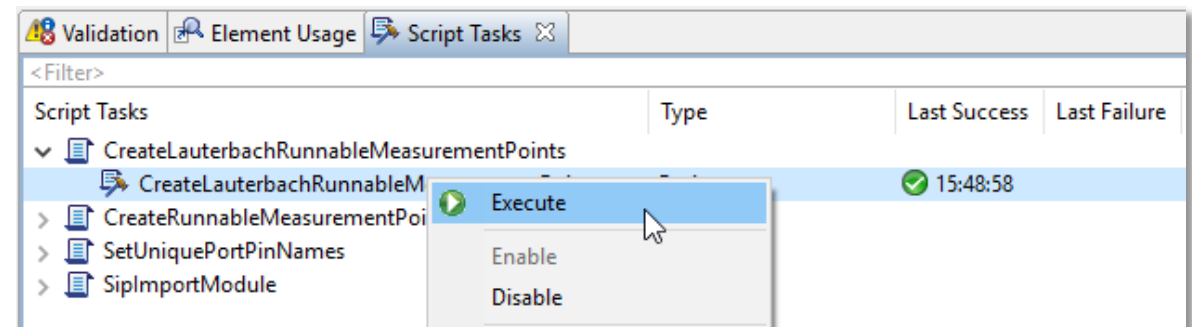
- ▶ To trace the start and the stop of Runnables, the VFB (Virtual Functional Bus) Tracing Functions can be activated
 - ▶ After RTE Generation the configured function are defined in the generated Rte_Hook.h



- ▶ By executing a DaVinci Automation Script called CreateLauterbachRunnableMeasurementPoints the hook functions are generated in an own C file (Rte_Hook_t32.c) and the Runnable mapping s generated in a CMM script (rte_runnable.cmm).



If the VFB Hooks are already in use, a new VFB Trace Client can be added. Please change therefore the VFB Hook Prefix filters in the python script.



Tracing Workflow

1.

Tracing Basic Knowledge

2.

Configure MICROSAR BSW – OS and RTE

3.

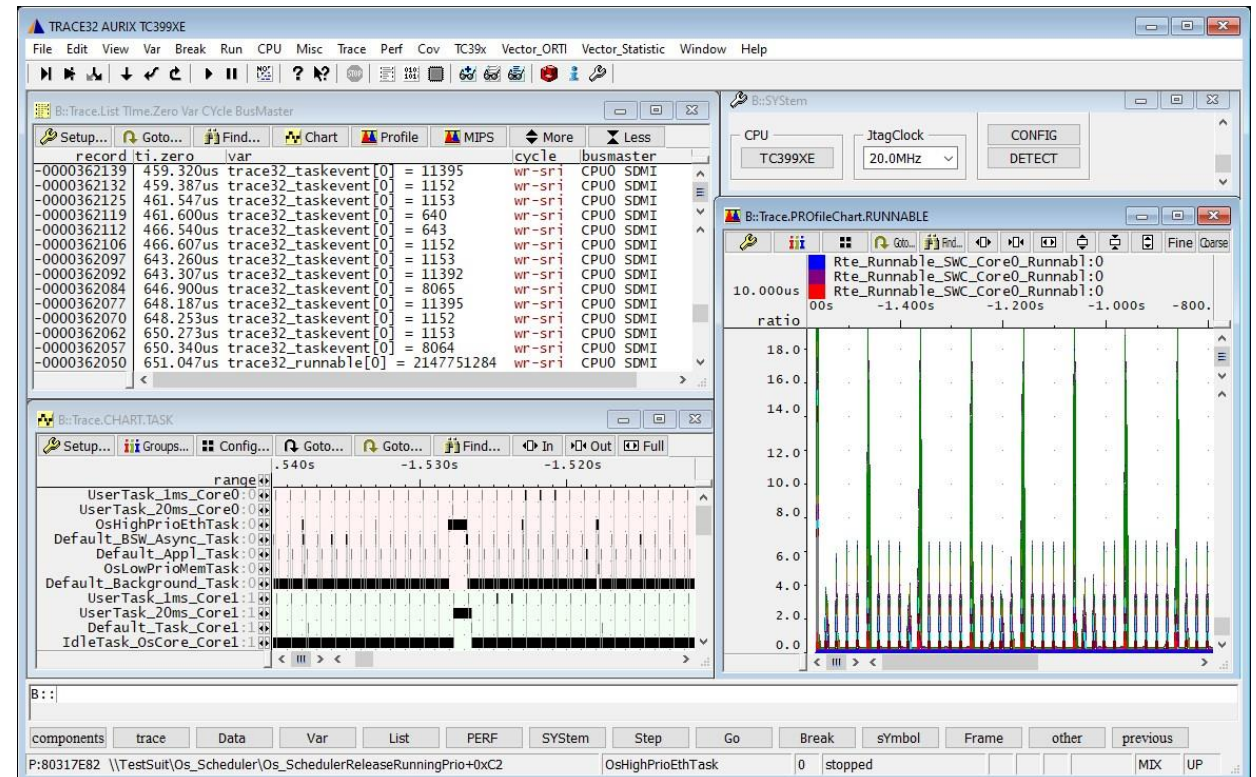
Configure Lauterbach Trace Debugger – TRACE32

4.

TA.Inspection - Trace Evaluation

Configure Lauterbach TRACE32

- ▶ To make the TRACE32 find all necessary OS information, load the ORTI file with the following command:
 - ▶ `TASK.ORTI Os_Trace.ORT`
- ▶ Set the tracing method according to your setup (Trace.METHOD)
- ▶ Call the script to configure the trace
 - ▶ for TriCore SRI Trace: `D0 vth_sri_trace.cmm`
 - ▶ for PowerPC Data Trace: `D0 vth_data_trace.cmm`
- ▶ Call the generated script to read the Runnable ID mapping (`rte_runnable.cmm`)



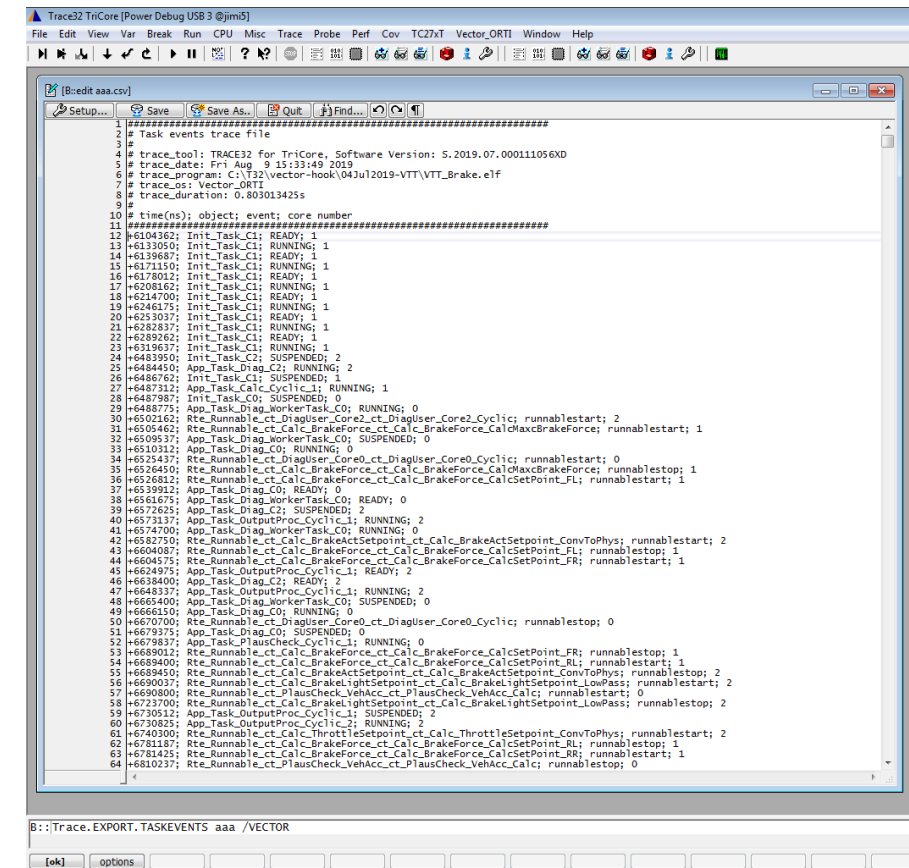
Configure Lauterbach TRACE32

- ▶ After hitting the Go button to start the target ECU, the tracing starts automatically
- ▶ With hitting the Break button, the measurement is stopped
- ▶ Now the scheduling trace can be exported with the following command:

▶ `Trace.EXPORT.TASKEVENTS <file_name.csv>`

or

▶ `Trace.EXPORT.MDF <file_name.MF4>`



The screenshot shows the TRACE32 software interface with a file named 'aaa.csv' open. The file contains a task events trace. The header section includes:

```
1 #####
2 # Task events trace file
3 #
4 # trace_tool: TRACE32 for TriCore, Software Version: 5.2019.07.000111056XD
5 # trace_date: Fri Jul 9 15:33:49 2019
6 # trace_program: C:\T32\vector-hook\04Jul2019-VTT\VT_Brake.elf
7 # trace_os: Vector_ORT1
8 # trace_duration: 0.803013425s
9 #
10 # time(ns); object; event; core number
11 #####
```

The main body of the file lists various tasks and their states, such as:

```
12 +6104362; Init_Task_C1; READY; 1
13 +6133050; Init_Task_C1; RUNNING; 1
14 +6139687; Init_Task_C1; READY; 1
15 +6171150; Init_Task_C1; RUNNING; 1
16 +6178012; Init_Task_C1; READY; 1
17 +6208162; Init_Task_C1; RUNNING; 1
18 +6214700; Init_Task_C1; READY; 1
19 +6246175; Init_Task_C1; RUNNING; 1
20 +6253037; Init_Task_C1; READY; 1
21 +6282837; Init_Task_C1; RUNNING; 1
22 +6289262; Init_Task_C1; READY; 1
23 +6319637; Init_Task_C1; RUNNING; 1
24 +6483950; Init_Task_C2; SUSPENDED; 2
25 +6484450; App_Task_Diag_C2; RUNNING; 2
26 +6486762; Init_Task_C1; SUSPENDED; 1
27 +6487312; App_Task_Calc_Cyclic1; RUNNING; 1
28 +6487987; Init_Task_C0; SUSPENDED; 0
29 +6488775; App_Task_Diag_WorkerTask_C0; RUNNING; 0
30 +6502162; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCMaxBrakeForce; runnablestart; 2
31 +6505462; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCMaxBrakeForce; runnablestart; 1
32 +6509937; App_Task_Diag_WorkerTask_C0; SUSPENDED; 0
33 +6510312; App_Task_Diag_C0; RUNNING; 0
34 +6525437; Rte_Runnable_ct_DiagUser_Core0_ct_DiagUser_Core0_Cyclic; runnablestart; 0
35 +6526450; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCMaxBrakeForce; runnablestop; 1
36 +6526812; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCSetPoint_FL; runnablestart; 1
37 +6539922; App_Task_Diag_C0; READY; 0
38 +6561675; App_Task_Diag_WorkerTask_C0; READY; 0
39 +6572625; App_Task_Diag_C2; SUSPENDED; 2
40 +6573137; App_Task_OutputProc_Cyclic1; RUNNING; 2
41 +6574700; App_Task_Diag_WorkerTask_C0; RUNNING; 0
42 +6582750; Rte_Runnable_ct_Calc_BrakeActSetpoint_ct_Calc_BrakeActSetpoint_ConvToPhys; runnablestart; 2
43 +6604087; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCSetPoint_FL; runnablestart; 1
44 +6604575; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCSetPoint_FR; runnablestart; 1
45 +6624975; App_Task_OutputProc_Cyclic1; READY; 2
46 +6638400; App_Task_Diag_C2; READY; 2
47 +6648337; App_Task_OutputProc_Cyclic1; RUNNING; 2
48 +6665400; App_Task_Diag_WorkerTask_C0; SUSPENDED; 0
49 +6666150; App_Task_Diag_C0; RUNNING; 0
50 +6670700; Rte_Runnable_ct_DiagUser_Core0_ct_DiagUser_Core0_Cyclic; runnablestop; 0
51 +6679375; App_Task_Diag_C0; SUSPENDED; 0
52 +6679837; App_Task_PlausCheck_Cyclic1; RUNNING; 0
53 +6689012; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCSetPoint_FR; runnablestop; 1
54 +6689400; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCSetPoint_FL; runnablestart; 1
55 +6689450; Rte_Runnable_ct_Calc_BrakeActSetpoint_ct_Calc_BrakeActSetpoint_ConvToPhys; runnablestop; 2
56 +6690037; Rte_Runnable_ct_Calc_BrakeLightSetpoint_ct_Calc_BrakeLightSetpoint_LowPass; runnablestart; 2
57 +6690000; Rte_Runnable_ct_PlausCheck_VehAcc_ct_PlausCheck_VehAcc_CalC; runnablestart; 0
58 +6723700; Rte_Runnable_ct_Calc_BrakeLightSetpoint_ct_Calc_BrakeLightSetpoint_LowPass; runnablestop; 2
59 +6730512; App_Task_OutputProc_Cyclic1; SUSPENDED; 2
60 +6730825; App_Task_OutputProc_Cyclic1; RUNNING; 2
61 +6740300; Rte_Runnable_ct_Calc_ThrottleSetpoint_ct_Calc_ThrottleSetpoint_ConvToPhys; runnablestart; 2
62 +6751187; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCSetPoint_FL; runnablestop; 1
63 +6781425; Rte_Runnable_ct_Calc_BrakeForce_ct_Calc_BrakeForce_CalCSetPoint_FR; runnablestart; 1
64 +6810237; Rte_Runnable_ct_PlausCheck_VehAcc_ct_PlausCheck_VehAcc_CalC; runnablestop; 0
```

At the bottom of the window, the command `B::Trace.EXPORT.TASKEVENTS aaa /VECTOR` is entered in the command line.

Tracing Workflow

1.

Tracing Basic Knowledge

2.

Configure MICROSAR BSW – OS and RTE

3.

Configure Lauterbach Trace Debugger – TRACE32

4.

TA.Inspection - Trace Evaluation

Benefits

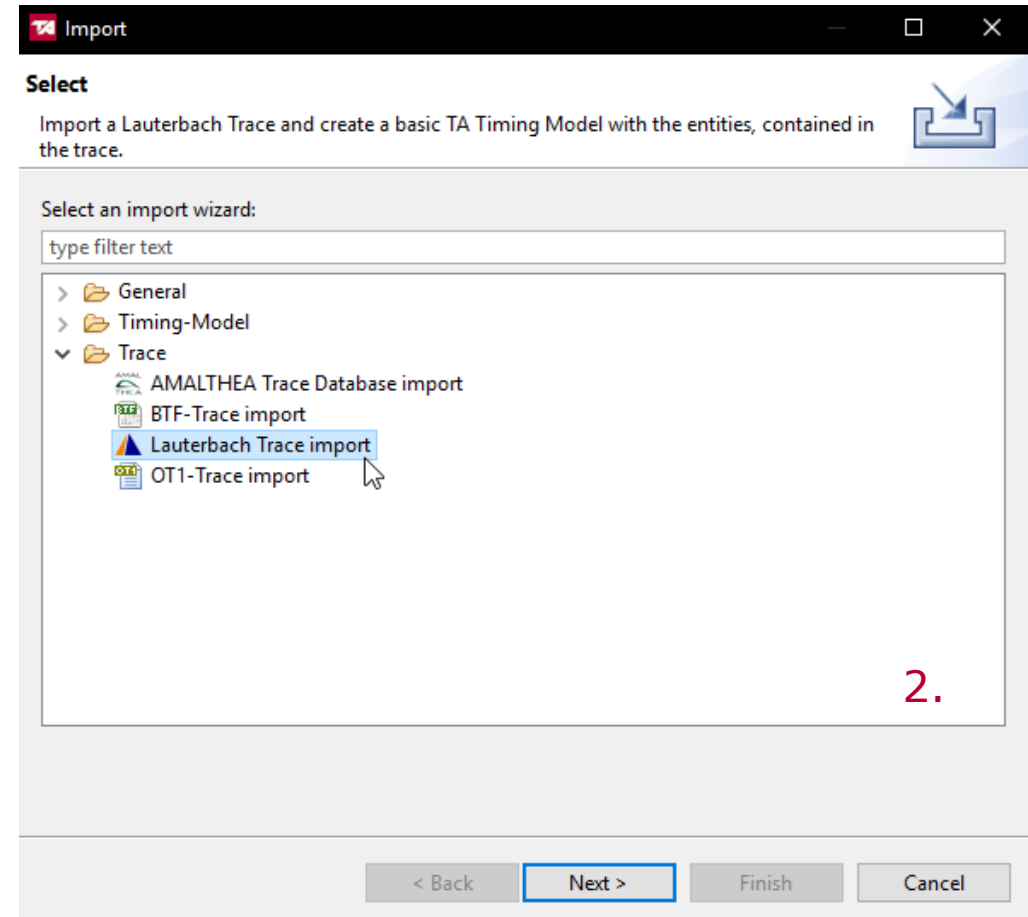
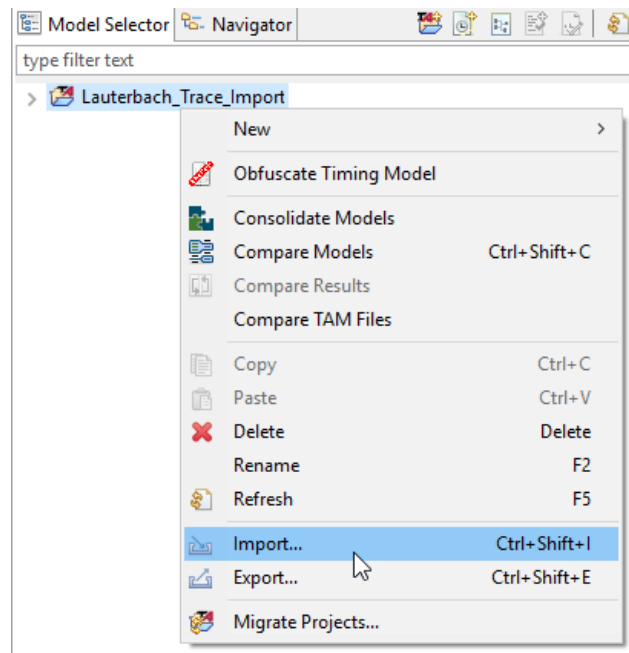
- ▶ Trouble shoot scheduling problems
- ▶ Test the timing behavior continuously and automatically
- ▶ Find sources of runtime consumption
- ▶ Evaluate the CPU Load and other metrics



Trace Import

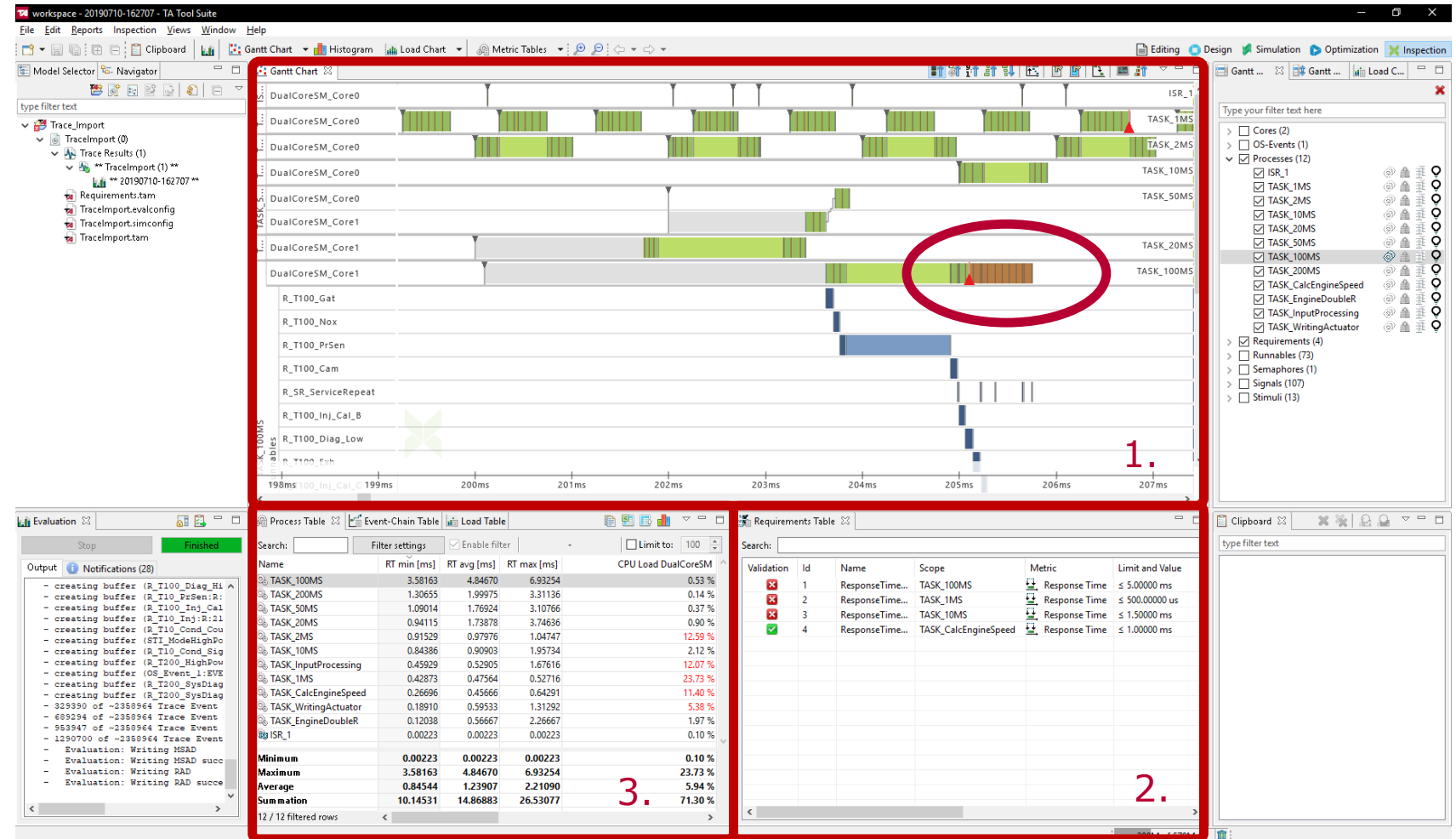
1. Right Click in the Model Selector View on Import
2. Select the Lauterbach Trace Import

1.



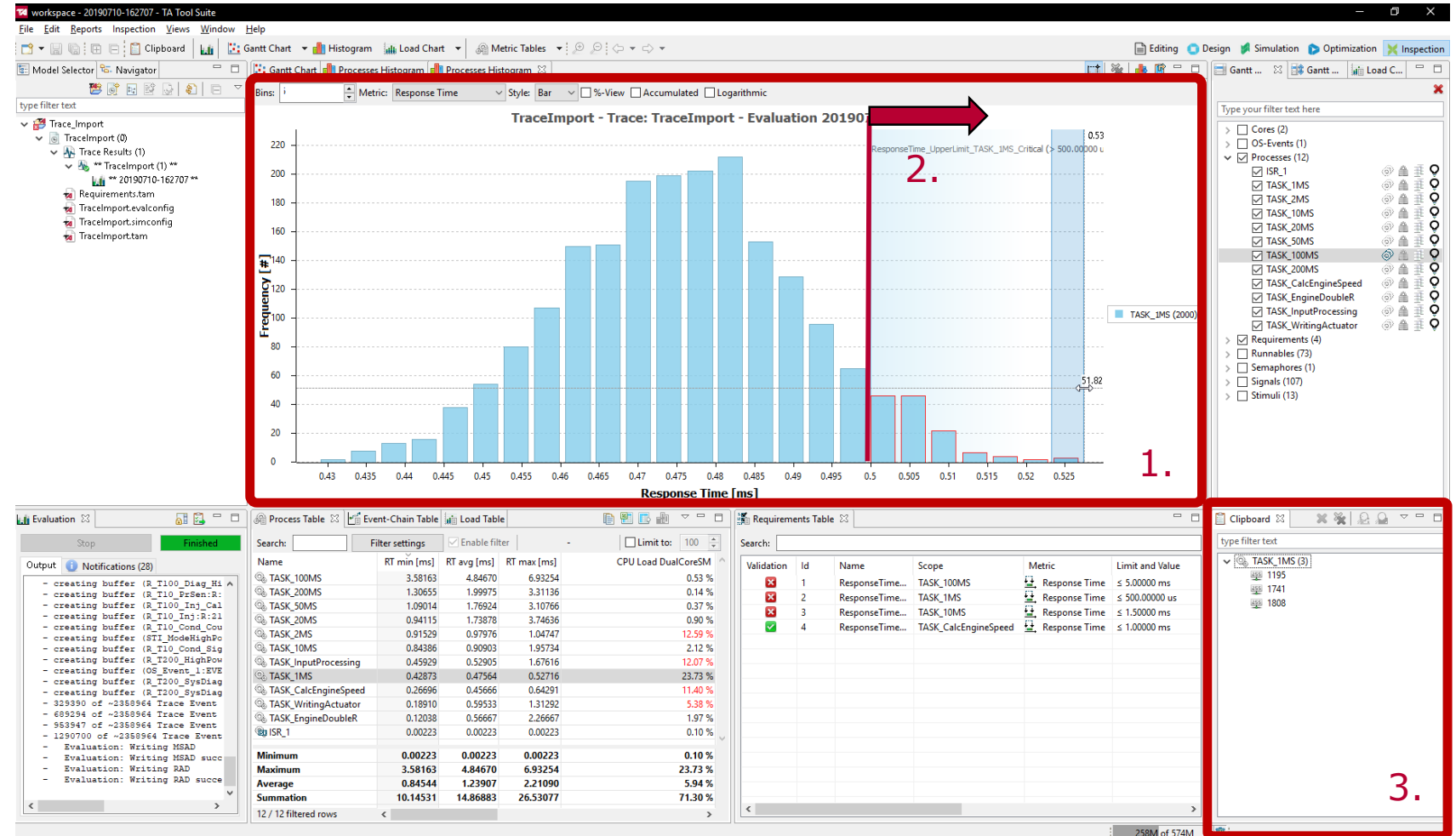
Troubleshoot the Runtime Behavior

1. Scheduling behavior
2. Requirements violations
3. Timing Metrics

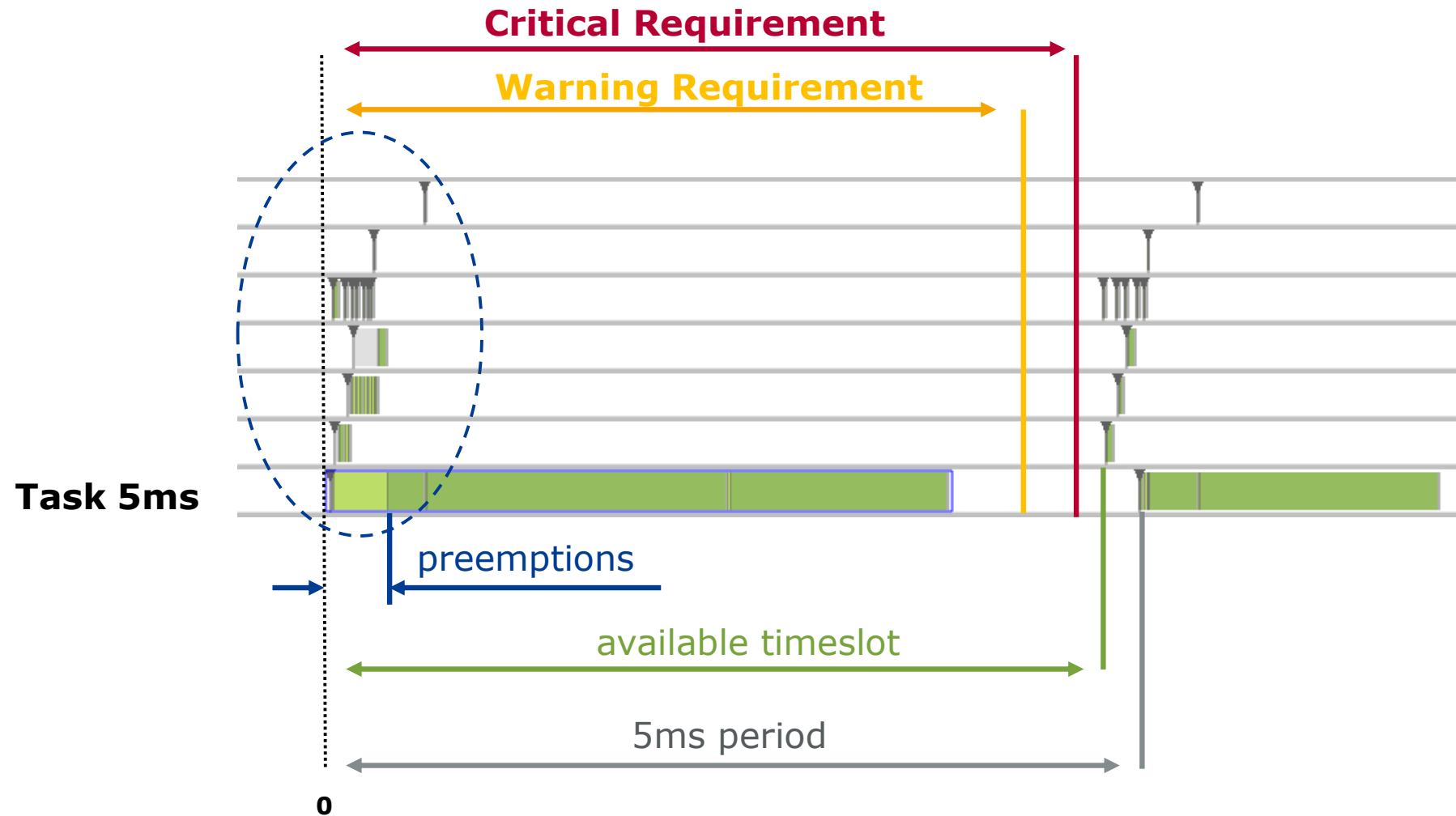


Analyze the Runtime Distribution

1. Analyze the Runtime Distribution
2. Requirements violations
3. Jump back to the Gantt to see the scheduling effects (previous slide 2.)



Derive Requirements



Report Generation and Export

- ▶ Document the Timing behavior throughout the development process
- ▶ Create Reports for customer requests
- ▶ Use the reports for internal audits



Customizable HTML Report

- ▶ Inputs such as Task Stack Consumption can also be stored



Evaluation Report

Contents

Input information
Overview
System Metrics
Task Metrics
ISR Metrics
Runnable Metrics
Appendix
Legend

Information

Information	Value
SVN Revision	34
Report creation	Mon Apr 15 09:34:16 CEST 2019
Evaluation date	2019-04-15T09:34:08Z
Evaluation path	
Evaluation requirements	
Evaluation requirements	
Evaluation tool	TA Eval 19.1.0.11
Accuracy	Displayed metrics might be rounded.

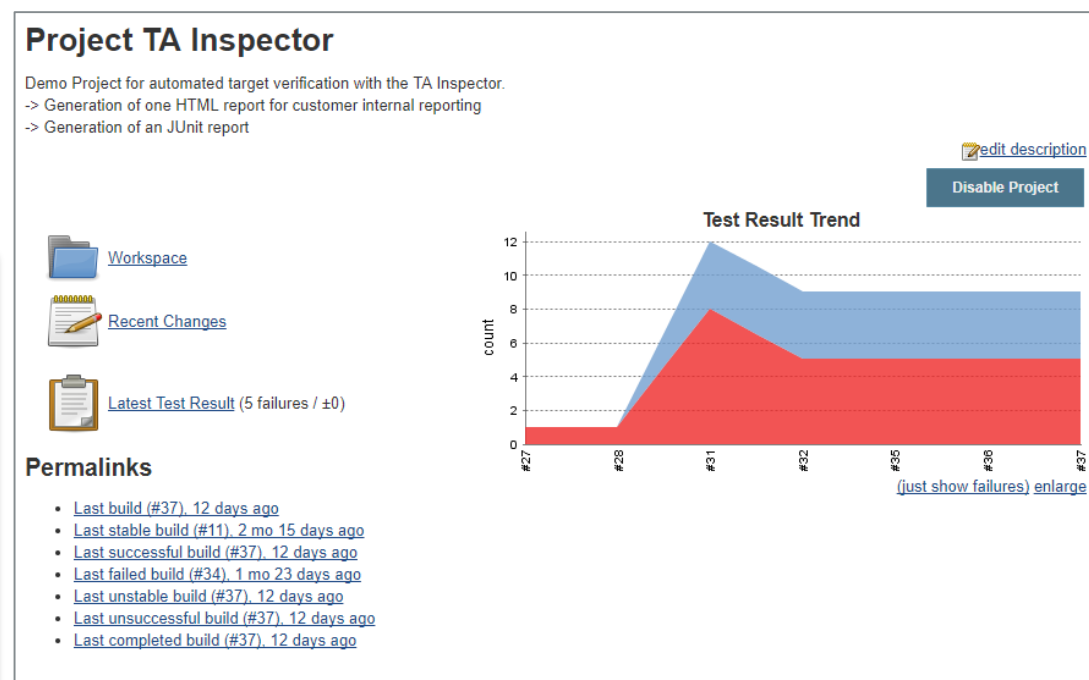
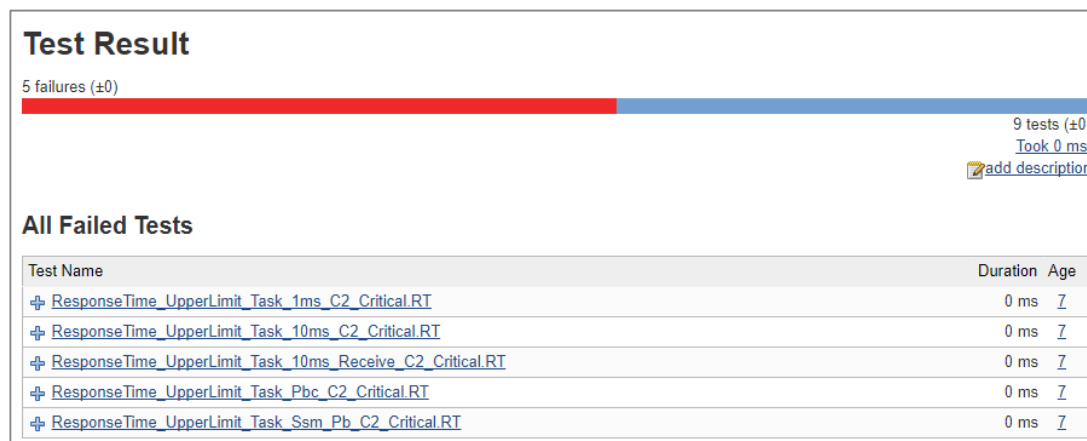
Requirements

Listing of all requirements to indicate their fulfillment

Entity	Metric	Severity	Threshold	Range	Occurrences [count]	Violations [count]	Violations [%]
	S2S [ms]	Critical Warning	11.0000 10.3000		153	0 0	0 0
	S2S [ms]	Critical Warning	55.0000 51.5000		29	0 0	0 0
	S2S [ms]	Critical Warning	5.5000 5.1500		307	0 1	0 0.326
	S2S [ms]	Critical Warning	11.0000 10.3000		153	0 10	0 6.536
	S2S [ms]	Critical Warning	110.0000 103.0000		14	0 0	0 0
	S2S [ms]	Critical Warning	11.0000 10.3000		154	0 0	0 0
	S2S [ms]	Critical Warning	110.0000 103.0000		14	0 0	0 0
	S2S [ms]	Critical Warning	11.0000 10.3000		153	0 2	0 1.307
	S2S [ms]	Critical Warning	5.5000 5.1500		304	17 124	5.592 40.789
	S2S [ms]	Critical Warning	22.0000 20.6000		75	0 14	0 18.667

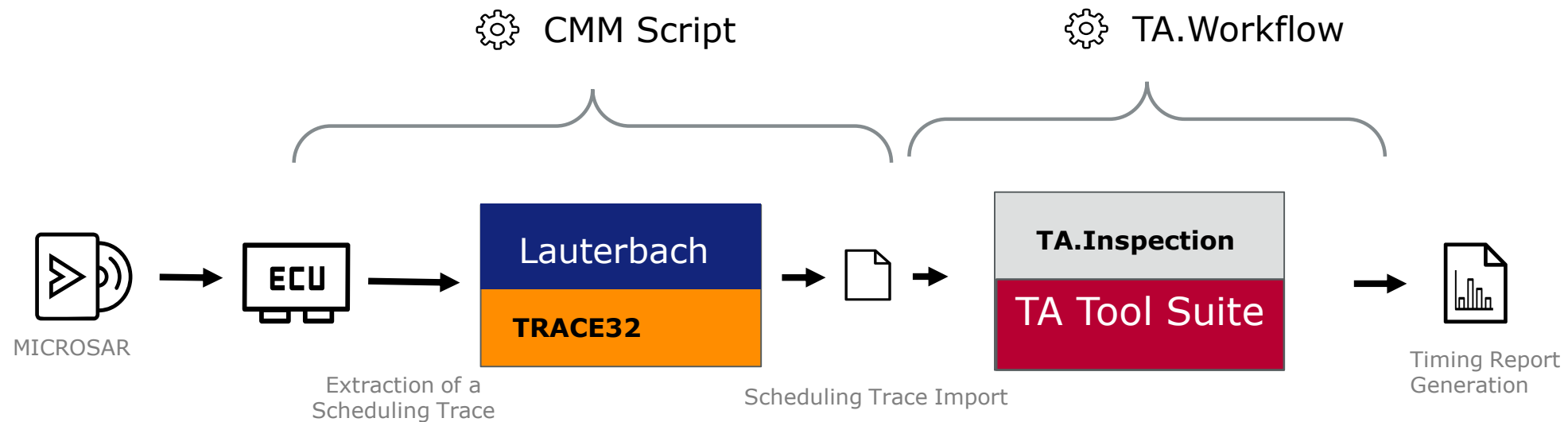
JUnit Reports and Continuous Timing Evaluation

- ▶ Junit Reports for automated testing
- ▶ Publish direct on the CI Server
- ▶ Track Timing Metrics continuously



Completely Automated Workflow

- ▶ The workflow can be completely automated:

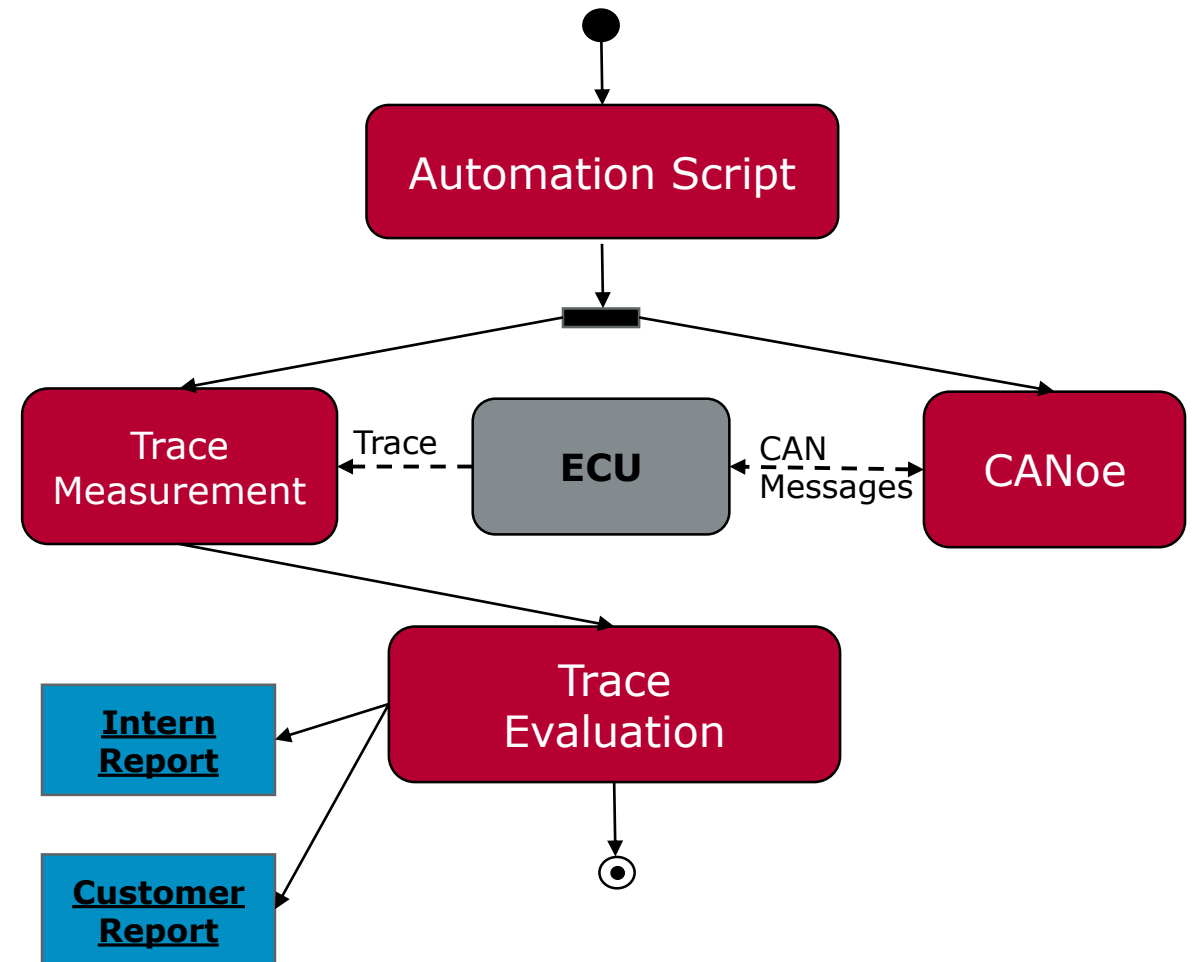


- ▶ Get the Timing results, with every build
- ▶ Publish the results in Jenkins as post build report



Tracing in ECU's Eco System with CANoe Automation

- ▶ Measure the ECU's runtime behavior in with external trigger
- ▶ Test ECU runtime performance in natural ECU environment
- ▶ Continuous testing throughout development process



For more information about Vector
and our products please visit

www.vector.com

Author:
Sebastian Ziegler
Vector Germany