MICROSAR RTE Analyzer

Technical Reference

Version 0.7.0

Authors	Sascha Sommer
Status	Released

Document Information

History

Author	Date	Version	Remarks
Sascha Sommer	2015-09-25	0.5	Initial creation for RTE Analyzer 0.5.0
Sascha Sommer	2016-02-26	0.6	Update for RTE Analyzer 0.6.0
Sascha Sommer	2016-07-07	0.7	Described Configuration Feedback and Template Variant Check

Reference Documents

No.	Source	Title	Version
[1]	ISO	ISO/IEC 9899:1990, Programming languages -C	Second edition
[2]	AUTOSAR	AUTOSAR_SWS_RTE.pdf	3.2.0

Scope of the Document

This technical reference describes the general use of the MICROSAR RTE Analyzer static code analysis tool. This document is relevant for developers that want to integrate a generated RTE into an ECU with functional safety requirements. All aspects that concern the generation of the RTE are described in the technical reference of the RTE.

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

1 **Component History** 5

2 **Introduction** 6

3 **Functional Description** 7
- 3.1 Features 7
- 3.2 Deviations 7

4 **RTE Analysis and Integration** 9
- 4.1 Scope of Delivery 9
  - 4.1.1 Static Files 9
  - 4.1.2 Dynamic Files 11
- 4.2 Restrictions 11
- 4.5 MicrosarRteAnalyzer.exe Command Line Options 12
- 4.6 Analysis Report Contents 12
  - 4.6.1 Analyzed Files 12
  - 4.6.2 Configuration Parameters 13
  - 4.6.3 Findings 13
  - 4.6.4 Configuration Feedback 20
  - 4.6.5 Template Variant Check 21
- 4.7 Integration into DaVinci CFG 22

5 **Glossary and Abbreviations** 25
- 5.1 Glossary 25
- 5.2 Abbreviations 25

6 **Additional Copyrights** 26

7 **Contact** 27

Tables

Table 1-1 Component history 5
Table 3-1 Supported features 7
Table 3-2 Not supported features 8
Table 4-1 Static files 10
Table 4-2 Generated files 11
Table 4-3 RTE Analyzer Command Line Options 12
Table 4-4 Analysis parameters that are extracted from the configuration 13
Table 4-5 RTE Analyzer Findings 20

Table 5-1 Glossary 25
Table 5-2 Abbreviations.. 25
Table 6-1 Free and Open Source Software Licenses 26

1 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
0.5.0	Initial version of MICROSAR RTE Analyzer for MICROSAR RTE 4.9.x Supported Features: - Detection of RTE code that cannot be compiled - Detection of Out Of Bounds write accesses within RTE APIs - Detection of Interrupt Lock API sequence mismatches within RTE APIs - Detection of Unreachable RTE APIs and runnables - Detection of RTE variables that are accessed from concurrent execution contexts without protection - Detection of concurrent calls to nonreentrant APIs within the RTE - Detection of variables that are accessed from multiple cores and that are not mapped to noncacheable memory sections - Detection of non typesafe interfaces to the BSW and SWCs where a call with a wrong parameter might cause out of bounds writes by the RTE or a called runnable/BSW API. - Detection of recursive call sequences
0.6.0	Updated for MICROSAR RTE 4.10.x
0.6.1	Updated for MICROSAR RTE 4.11.x
0.7.0	Updated for MICROSAR RTE 4.12.x New optimized Range Analysis algorithm Added Configuration Feedback Added Template Variant Check

Table 1-1 Component history

2 Introduction

This document describes the static code analysis tool MICROSAR RTE Analyzer. MICROSAR RTE Analyzer is part of MICROSAR Safe RTE. MICROSAR Safe RTE provides an AUTOSAR RTE generator that is developed with an ISO26262 compliant development process, to allow the usage of the generated RTE code within an ECU with functional safety requirements.

MICROSAR RTE Analyzer analyzes the generated RTE code for errors with a special emphasis on sporadic runtime errors that are hard to detect during ECU integration tests.

Caution

This version of MICROSAR RTE Analyzer is a preview version. While many of the errors that are described in the feature list can be detected, the development and certification of MICROSAR RTE Analyzer and MICROSAR Safe RTE are still in the works.

The usage of this version of MICROSAR RTE Analyzer alone is therefore no sufficient prove that the generated RTE can be used in an ECU with functional safety requirements.

3 Functional Description

This chapter describes how the tool MICROSAR RTE Analyzer can be used to check the consistency of the generated RTE.

3.1 Features

The features listed in the following table cover the complete functionality of MICROSAR RTE Analyzer.

Supported Features
Compilation check for RTE code
Detection of Out Of Bounds write accesses within RTE APIs
Detection of Interrupt Lock API sequence mismatches within RTE APIs
Detection of Unreachable RTE APIs and runnables
Detection of RTE variables that are accessed from concurrent execution contexts without protection
Detection of concurrent calls to nonreentrant APIs within the RTE
Detection of variables that are accessed from multiple cores and that are not mapped to noncacheable memory sections
Detection of non typesafe interfaces to the BSW and SWCs where a call with a wrong parameter might cause out of bounds writes by the RTE or a called runnable/BSW API
Detection of RTE APIs for which a call from a wrong context might cause data consistency problems
Detection of recursive call sequences
Analysis report generation
Configuration Feedback Generation
Template Variant Check

Table 3-1 Supported features

3.2 Deviations

The following features are not yet supported:

Not Supported Features
Configuration Feedback for implicit exclusive areas
Configuration Feedback for implicit inter runnable variables
Configuration Feedback for implicit APIs with data conversion
Configuration Feedback for implicit APIs with E2E protection
Configuration Feedback for Inter-ECU client-/server communication with E2E protection
Configuration Feedback for DiagXf
Automatic verification of COM buffer assumptions for MICROSAR COM

Not Supported Features
Automatic verification of NVM buffer assumptions for MICROSAR NVM

Table 3-2 Not supported features

4 RTE Analysis and Integration

This chapter gives necessary information about the content of the delivery, the usage of MICROSAR RTE Analyzer and a description of the generated report.

4.1 Scope of Delivery

The delivery contains the files which are described in the chapters 4.1.1 and 4.1.2:

4.1.1 Static Files

File Name	Description
MicrosarRteAnalyzer.exe	MICROSAR RTE Analyzer commandline frontend
MicrosarRteAnalyzerCfgGen.exe	MICROSAR RTE Analyzer configuration file generator (automatically invoked by DaVinci CFG during RTE generation)
Settings_RteAnalyzer.xml	Davinci CFG adaption module
TechnicalReference_RteAnalyzer.pdf	This document
clang.exe	CLANG compiler frontend (used internally by MicrosarRteAnalyzer.exe)
llvm-link.exe	LLVM linker (used internally by MicrosarRteAnalyzer.exe)
MicrosarIRAnalyzer.exe	Analysis backend (used internally by MicrosarRteAnalyzer.exe)
License_Artistic.txt	Perl license
License_LLVM.txt	LLVM/CLANG license
Com.h	Stub Com header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Compiler.h	Stub Compiler header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Compiler_Cfg.h	Stub Compiler_Cfg header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
ComStack_Cfg.h	Stub ComStack_Cfg header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
ComStack_Types.h	Stub ComStack_Types header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Det.h	Stub Det header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
E2EXf.h	Stub E2EXf header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)

File Name	Description
Float.h	Stub Float header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Ioc.h	Stub Ioc header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
LdCom.h	Stub LdCom header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
MemMap.h	Stub MemMap header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
NvM.h	Stub NvM header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Os.h	Stub Os header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Os_MemMap.h	Stub Os_MemMap header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Platform_Types.h	Stub Platform_Types header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Std_Types.h	Stub Std_Types header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
String.h	Stub String header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
Xcp.h	Stub Xcp header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
XcpProf.h	Stub XcpProf header (used internally by MicrosarRteAnalyzer.exe for standalone RTE verification)
demo	Example RTE with problems that can be found by RTE Analyzer.

Table 4-1 Static files

4.1.2 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci CFG to the RteAnalyzer subdirectory when the RTE is generated.

File Name	Description
RteAnalyzerConfiguration.json	Configuration file for MICROSAR RTE Analyzer.
<BSW>.c	These files contain stub implementations for the schedulable entities that call all available RTE APIs.
TestControl.c	This file contains stubs for the BSW calls to the RTE.
<SWC>.c	These files contain stub implementations for the runnables that call all available RTE APIs.
Com_Cfg.h	This file contains the configuration for the stub COM module.
LdCom_Cfg.h	This file contains the configuration for the stub LDCOM module.
Os_Cfg.h	This file contains the configuration for the stub OS module.
Ioc_Cfg.h	This file contains the IOC configuration for the stub OS module.
Xcp_Cfg.h	This file contains the configuration for the stub XCP module.
NvM_Cfg.h	This file contains the configuration for the stub NVM module.
E2EXf_Cfg.h	This file contains the configuration for the stub E2E module.

Table 4-2 Generated files

Besides the files that are generated by DaVinci CFG, RTE Analyzer generates the following files when invoked from the commandline.

File Name	Description
AnalysisReport.txt	Report that contains the results of the static code analysis and analysis assumptions that need to be reviewed by the user of MICROSAR RTE Analyzer.

4.2 Restrictions

MICROSAR RTE Analyzer uses a Compiler front end in order to compile the input source files. This Compiler front end requires ANSI-C 90 [1] conform source code. Some target compilers implement specific language extensions which might prevent MICROSAR RTE Analyzer from compiling the code successfully. The Vector BSW code does not contain such language extensions. However, these extensions may be included via customer header files. In such a case the customer shall take care that these language extensions are encapsulated via the preprocessor for the MICROSAR RTE Analyzer execution. The corresponding preprocessor switches can be specified via the command line when calling MICROSAR RTE Analyzer.

4.5 MicrosarRteAnalyzer.exe Command Line Options

The frontend MicrosarRteAnalyzer.exe starts the static code analysis. It can be started on the commandline once the RTE and the MICROSAR RTE Analyzer configuration were generated by DaVinci CFG.

Option	Description
`-c <config>`	Selects the configuration file of the project that shall be analyzed.
`-I <dir>`	Add directory name <dir> to include file search path
`-D <name>[=<value>]`	Defines macro with name <name> and value <value>
`-o <path>`	Selects the directory to which the analysis report will be written
`-e`	Extended Configuration Feedback. If not set, the Configuration Feedback will not include RTE functionality in OS Applications with SafetyLevel QM.
`-V`	Shows the version
`-h`	Shows the commandline help

Table 4-3 RTE Analyzer Command Line Options

Example:

```
MicrosarRteAnalyzer.exe -c RteAnalyzerConfiguration.json -o
Reports
```

A small example RTE and RTE Analyzer configuration is contained in the subfolder demo.

4.6 Analysis Report Contents

MicrosarRteAnalyzer.exe prints errors that prevent the analysis of the system to the console.

When MicrosarRteAnalyzer.exe was executed without errors, an analysis report is written to the output directory that contains potential problems within the generated RTE.

These problems are only listed in the report and not printed to the console.

As not every detected violation necessarily leads to an error in the ECU, the final decision whether an issue is critical or not is up to the user of MicrosarRteAnalyzer.exe.

Besides the detected constraint violations, the analysis report also contains assumptions about the system that were derived from the configuration.

These assumptions need to be verified by the user of MicrosarRteAnalyzer.exe.

4.6.1 Analyzed Files

The report starts with the version of the analysis report, the time of the analysis and the name of the windows user that initiated the analysis.

Moreover the analyzed files are listed. It needs to be assured that the correct files were analyzed and no file is missing.

4.6.2 Configuration Parameters

MICROSAR RTE Analyzer relies on configuration parameters from DaVinci CFG to determine the scheduling properties of the individual tasks and BSW callbacks.

These parameters need to be reviewed because a wrong parameter might lead to missed data consistency problems.

The report contains the following parameters that need to be checked against the target system.

Parameter	Description
MaxAtomicMemoryAccess	Describes the maximum number of bytes for variable accesses up to which the compiler will emit an atomic access instruction.
BswOsApplication	Describes the OS Application from which the RTE Callbacks (Rte_COMCbk, Rte_LdCom, Rte_GetMirror, Rte_SetMirrors, …) are called.
OsApplications	Lists the OS Applications in the system
OsApplicationName	Name of the OS Application
CoreId	ID of the Core that contains the OS Application
IsTrusted	Describes if the OS Application runs without MPU (IsTrusted == 1) or with MPU (IsTrusted == 0)
SafetyLevel	SafetyLevel of the OS Application: QM, ASIL_A, ASIL_B, ASIL_C, ASIL_D
Tasks	List of OS Tasks that are assigned to the OS Application
TaskName	Name of the OS Task
Priority	Priority of the OS Task
Preemption	Preemption setting of the OS Task

Table 4-4 Analysis parameters that are extracted from the configuration

4.6.3 Findings

RTE Analyzer currently reports the findings described in Table 4-5. The description describes the possible findings in more detail and the actions that need to be taken when they are contained in the analysis report.

ID	Headline	Description
11000	Unsupported integer to pointer conversion	RTE code uses an integer value that was

ID	Headline	Description
		casted to a pointer type. Example: *uint8* ptr = 0xdeadbeef;* **ptr = 5;* This code construct must not be used in the RTE code. Contact Vector.
11001	Unsupported inline assembly	RTE code uses inline assembly. Example: *asm("add %al, (%rax)");* This code construct must not be used in the RTE code. Contact Vector.
11002	Function may be called recursively	The software design contains e.g. configured client server calls that may lead to recursive calls. ISO26262 recommends that recursion is not used in the software design and implementation.
11003	Uncalled function	A function e.g. a server runnable without connected client was encountered during the analysis. Functions that are not called are not analyzed by RTE Analyzer. Assure that the function is not called in the target system, either.
11004	Uncalled function (reachable from other uncalled function)	A function that is unreachable because it is only called by other uncalled functions e.g. a server runnable without connected client was encountered during the analysis. Unreachable functions that are not called are not analyzed by RTE Analyzer. Assure that the function is not called in the target system, either.
11005	Library function with unhandled side effects	RTE code calls a library function with a pointer parameter that is not known to RTE Analyzer. The function may have unexpected side effects. Contact Vector.
11006	Unsupported path to pointer target	The pointer analysis detected a code construct that it cannot handle. This code construct must not be used in RTE code. Contact Vector.
12000	Potential out of bounds write	A pointer that was already used in the preparation of the analysis is outside of the assumptions that were used during the preparations. Example:

ID	Headline	Description
		typedef struct { *uint8* a;* *uint8* b;* *} struct_t;* *struct_t s;* *uint8** ptr = &s.a;* *ptr[1][0] = 7;* This code construct must not be used in the RTE code. Contact Vector.
12001	Potential null pointer write	An RTE API writes to a pointer that may be null. This code construct must not be used in the RTE code. Contact Vector.
12002	Potential out of bounds write	An RTE API writes outside of the bounds of a variable. Example: *uint8 a[5];* *a[5] = 1;* This code construct must not be used in the RTE code. Contact Vector.
13000	Unexpected lock sequence	A lock function is not followed by an appropriate unlock function. Example: *SuspendAllInterrupts();* *a = 5;* *ResumeOSInterrupts();* This code construct must not be used in the RTE code. Contact Vector.
13001	Different lock states for loop	A function uses different lock states in different loop iterations. Example: *for (i = 0; i < 20; i++)* *{* *if (i == 5)* *{* *DisableAllInterupts();* *}*

ID	Headline	Description
		if (i == 6) *{* *EnableAllInterrupts();* *}* *}* This code construct must not be used in the RTE code. Contact Vector.
13002	Different lock states for call	A call may be done with and without prior locking. Example: *if (a == 0)* *{* *DisableAllInterrupts();* *}* *Function();* This code construct must not be used in the RTE code. Contact Vector.
13003	Different lock states for recursive call	A recursive function changes the lock state prior to the next recursion. Example: *void func()* *{* *DisableAllInterrupts();* *func();* *EnableAllInterrupts();* *}* This code construct must not be used in the RTE code. Contact Vector.
13004	Different lock states for return	A function may return a with different lock state. Example: *DisableAllInterrupts();* *if (a)* *{* *return;* *}* *EnableAllInterrupts();* *return;*

ID	Headline	Description
		This code construct must not be used in the RTE code. Contact Vector.
13005	Task or ISR returns with locked interrupts	A RTE Task or callback returns with locked interrupts in at least one code branch. This code construct must not be used in the RTE code. Contact Vector.
13006	OS API called with locked interrupt	An OS API e.g. WaitEvent is called with locked interrupts. This is prohibited by the OS specification. This code construct must not be used in the RTE code. Contact Vector.
13007	OS API called with disabled interrupts	An OS API e.g. SuspendOSInterrupts is called within a section that is locked with DisableAllInterrupts. This is prohibited by the OS specification. This code construct must not be used in the RTE code. Contact Vector.
13008	OS API called in wrong context	An optimized MICROSAR interrupt lock API is called from the wrong context. E.g. an optimized lock API for trusted OS application is called from an untrusted application. This code construct must not be used in the RTE code. Contact Vector.
13009	Accesses can interrupt each other	RTE Analyzer detected that a variable is accessed from multiple tasks that can interrupt each other. The variable is not protected by an OS API e.g. interrupt lock or spinlock. This code construct must not be used in the RTE code. Contact Vector.
13010	Nonreentrant function with nonconstant handle	RTE Analyzer checks the RTE for concurrent calls to BSW APIs. If the reentrancy depends on the handle, the handle needs to be constant so that it can be analyzed by RTE Analyzer. This code construct must not be used in the RTE code. Contact Vector.
13011	Nonreentrant function invoked concurrently	RTE Analyzer detects concurrently called functions for which the caller would have needed to assure nonreentrant calls. This code construct must not be used in the RTE code. Contact Vector.
13012	Different resources used on same core	The RTE code uses different resources to protect the same variable. If a variable needs to be protected from concurrent accesses in multiple tasks, the same

ID	Headline	Description
		resource needs to be used for all accesses. This code construct must not be used in the RTE code. Contact Vector.
13013	Different spinlocks used	The RTE code uses different spinlocks to protect the same variable on a single core. If a variable needs to be protected from concurrent accesses on multiple cores, the same spinlock needs to be used for all accesses. This code construct must not be used in the RTE code. Contact Vector.
13014	Not all accesses protected with resource	The RTE code does not always use resources to protect a variable. If a variable needs to be protected from concurrent accesses in multiple tasks, the same resource needs to be used for all accesses. This code construct must not be used in the RTE code. Contact Vector.
13015	Bitfield write access without interrupt locks	The RTE uses interrupt locks to prevent read modify write problems in bitfields. RTE Analyzer detected an access without locks. This code construct must not be used in the RTE code. Contact Vector.
14000	Unmatched memory section	A memory section was not closed correctly. Example: *#define RTE_START_SEC_VAR* *#include “MemMap.h”* *uint8 var;* *#define RTE_STOP_SEC_CONST* *#include “MemMap.h”* This code construct must not be used in the RTE code. Contact Vector.
14001	Variable not mapped to memory section	A variable is declared without being mapped to a memory section. Example: *uint8 var;* *#define RTE_START_SEC_VAR* *#include “MemMap.h”* This code construct must not be used in the RTE code. Contact Vector.
14002	Variable not mapped to NOCACHE	A variable that is accessed from multiple

ID	Headline	Description
	memory section	cores is not mapped to a NOCACHE memory section. This may lead to data consistency problems. Example: *#define RTE_START_SEC_VAR* *#include “MemMap.h”* *uint8 var;* *#define RTE_STOP_SEC_VAR* *#include “MemMap.h”* This code construct must not be used in the RTE code. Contact Vector.
15000	Call with non typesafe parameters	Some APIs contain pointers that are not typesafe e.g. because the parameter type is a pointer to the base type and the function writes more than a single element of this type. Example: *void Rte_API(uint8* a)* *{* *a[2] = 4;* *}* *void Runnable()* *{* *uint8 a[3];* *Rte_API(&a[0]);* *}* The parameter may also be a void pointer type. RTE Analyzer lists these functions so that the user can verify that the passed buffer matches the expectations of the called function. Please note that the buffer that is listed by RteAnalyzer might be larger than the actual number of bytes that are written by the called function.
15001	API for Safe component must not be called from wrong context	The RTE generator disables task priority optimizations for partitions with an ASIL Safety Level. If an API is used only on a single task according to the configuration, the RTE generator optimizes

ID	Headline	Description
		nevertheless. RTE Analyzer lists these APIs so that the user can verify that he does not accidently call the API from a runnable for which no port access was configured in the configuration.
16000	Missing task info	The configuration contains no task settings for the task. Possible reason: a function ends with the name func and is missdetected as OS Task by RTE Analyzer. Rename the function or ignore the message.

Table 4-5 RTE Analyzer Findings

4.6.4 Configuration Feedback

The findings from chapter 4.6.3 describe inconsistencies within the generated RTE. However, also a consistently generated RTE may violate functional safety requirements when the generated RTE does not match the intentions of the user e.g. when wrong configuration parameters were chosen for the intended use case.

Therefore, during development of the RTE Generator a safety analysis is performed on all input parameters of the generator in order to detect functionality for which a slightly different configuration leads to the generation of APIs with compatible C signature but different runtime behavior.

RTE Analyzer lists the detected functionality in the analysis report, so that an integration test can confirm that the intended functionality is implemented in the generated RTE.

This also makes it possible to use MICROSAR RTE Generator in combination with non TCL1 configuration tools as unintended configuration modifications by the tool will lead to an unexpected configuration feedback.

By default the configuration feedback is only printed for the OS Applications with ASIL safety levels. When the –e configuration switch is enabled, the RTE functionality in OS Applications with SafetyLevel QM is also included. Analysis report contains the following information:

- Non-Queued connections – This contains list of all non-queued intra-ECU sender-receiver connections between Rte_Write, Rte_IWrite, Rte_Read, Rte_DRead, Rte_IRead.
- Queued connections – This contains list of all queued intra-ECU sender-receiver communication connections between Rte_Send and Rte_Receive.
- Inter-runnable connections – This contains list of all inter-runnable variable connections.
- External connections – This contains list of all the APIs and server runnables that communicate with other ECUs.

- Switch-mode connections – This contains list of all mode connections between Rte_Switch and Rte_Mode.
- Exclusive areas – This contains list of all exclusive areas and their implementation methods. This includes explicit and implicit exclusive areas. The implementation methods need to be set according to the requirements of the application.
- Initial values of APIs – This contains list of all the APIs that return an initial value. The calling runnable needs to handle the initial value. When RteAnalyzer was able to extract the initial value from the code, the value is also printed.
- Blocking APIs – This contains list of all APIs that are blocking. These may unexpectedly delay the calling function.
- Executable Entities – This contains list of all the executable entities. The entities are listed together with the tasks in which they are executed.
- APIs with special return values – This contains list of all the APIs that return special error codes such as RTE_E_MAX_AGE_EXCEEDED, RTE_E_INVALID and RTE_E_NEVER_RECEIVED.
- APIs with queues – This contains the list of APIs with queues along with the queue sizes.
- APIs with E2E transformers – This contains the list of APIs that read or write data with the help of the E2E transformer. The communication partner needs to handle the converted data.
- Reentrant Executable Entities – This contains list of all executable entities that are called reentrantly. This is based on the core id, priority and the preemption setting of the tasks in which the entity is executed.
- APIs using data conversion – This contains list of all the APIs that do data conversion. The communication partner needs to handle the converted data.
- APIs that may use NVM – This contains list of all Per Instance Memories and sender-receiver APIs that access NV Block SWCs. The NVM module needs to be configured correctly.

Please note that the configuration feedback describes the actual properties of the code. This can be different from the configured values, especially if the APIs are generated for unconnected ports.

Example: An unconnected Rte_Read API is configured to return RTE_E_NEVER_RECEIVED. According to the RTE specification, the return value is RTE_E_UNCONNECTED independently of the never received handling, therefore the generated API has no code to return RTE_E_NEVER_RECEIVED and the analysis report does not list the API in the “APIs with special return values” section.

The safety manual describes how the configuration feedback can be used for integration testing.

4.6.5 Template Variant Check

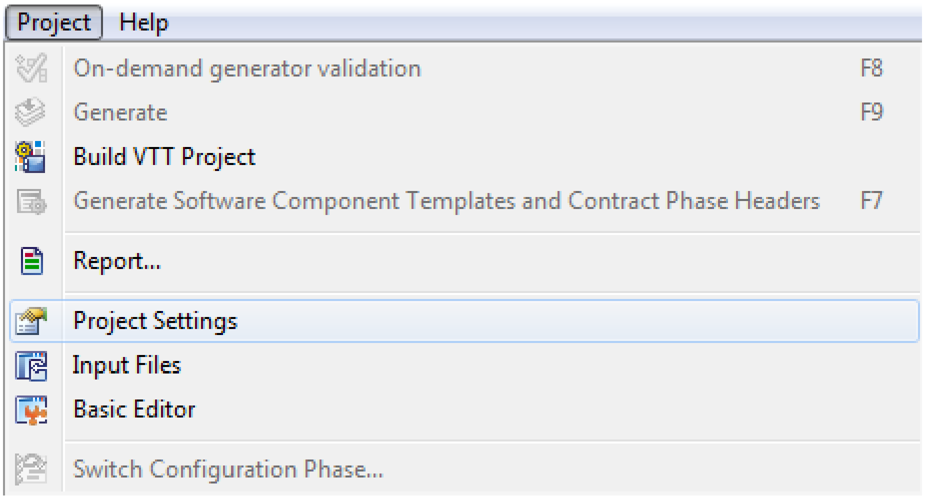
MICROSAR RTE Generator is a template based code generator. During generation, MICROSAR RTE Generator calculates checksums for the template sequences that were used to generate the RTE APIs. The delivery of the generator contains a list of checksums that were approved for the usage in an ECU with functional safety requirements.

MICROSAR RTE Analyzer checks that the template sequences that were used to generate the analyzed RTE are within the allowed sequences.

Please contact Vector if the analysis report lists template variants that are not within the allowed ones.

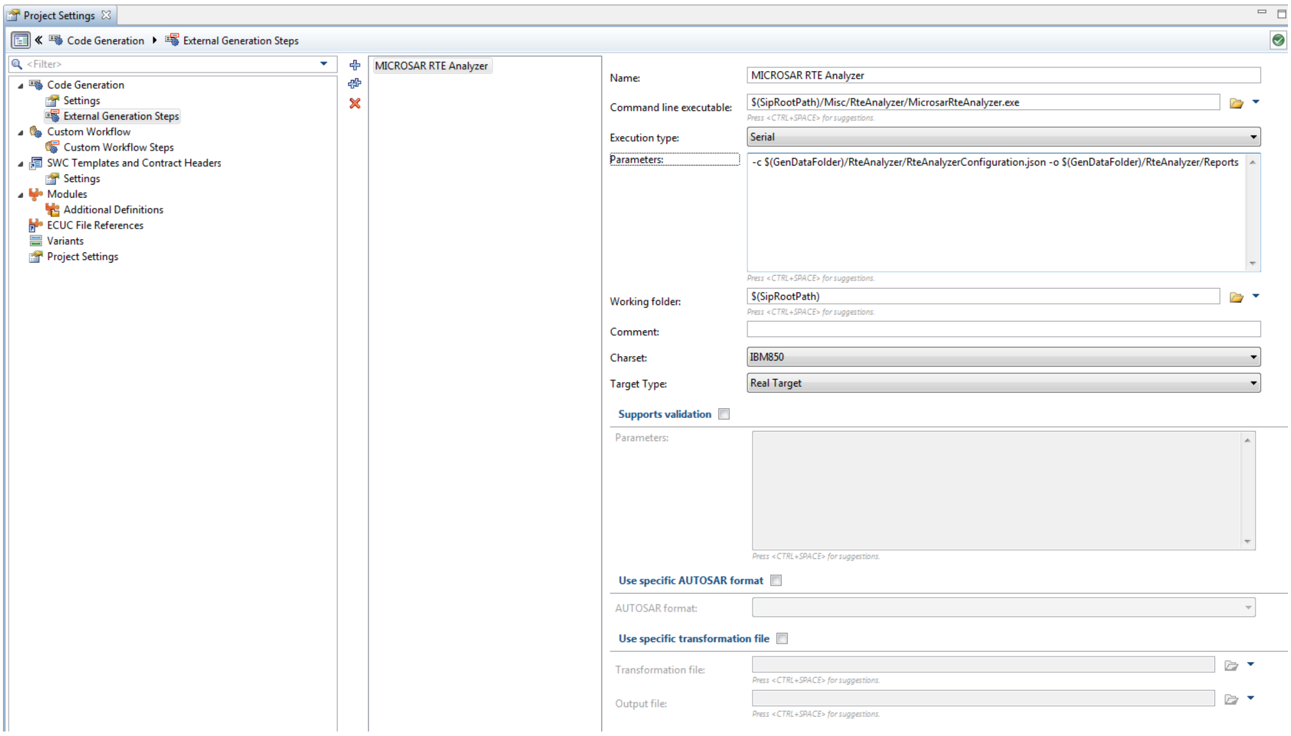
4.7 Integration into DaVinci CFG

Since MICROSAR RTE Analyzer checks the consistency of the generated RTE it is convenient to run MICROSAR RTE Analyzer automatically after the data is generated. To integrate MICROSAR RTE Analyzer into DaVinci CFG, an external generation step can be configured.



Start DaVinci CFG and select the menu “Project”. Next select the menu item “Settings”.

To add a new external generation step, select “External Generation Steps”. This will display the following window:



Click on the Add button with the “+” symbol and enter the MICROSAR RTE Analyzer path e.g.

`$(SipRootPath)/Misc/RteAnalyzer/MicrosarRteAnalyzer.exe`

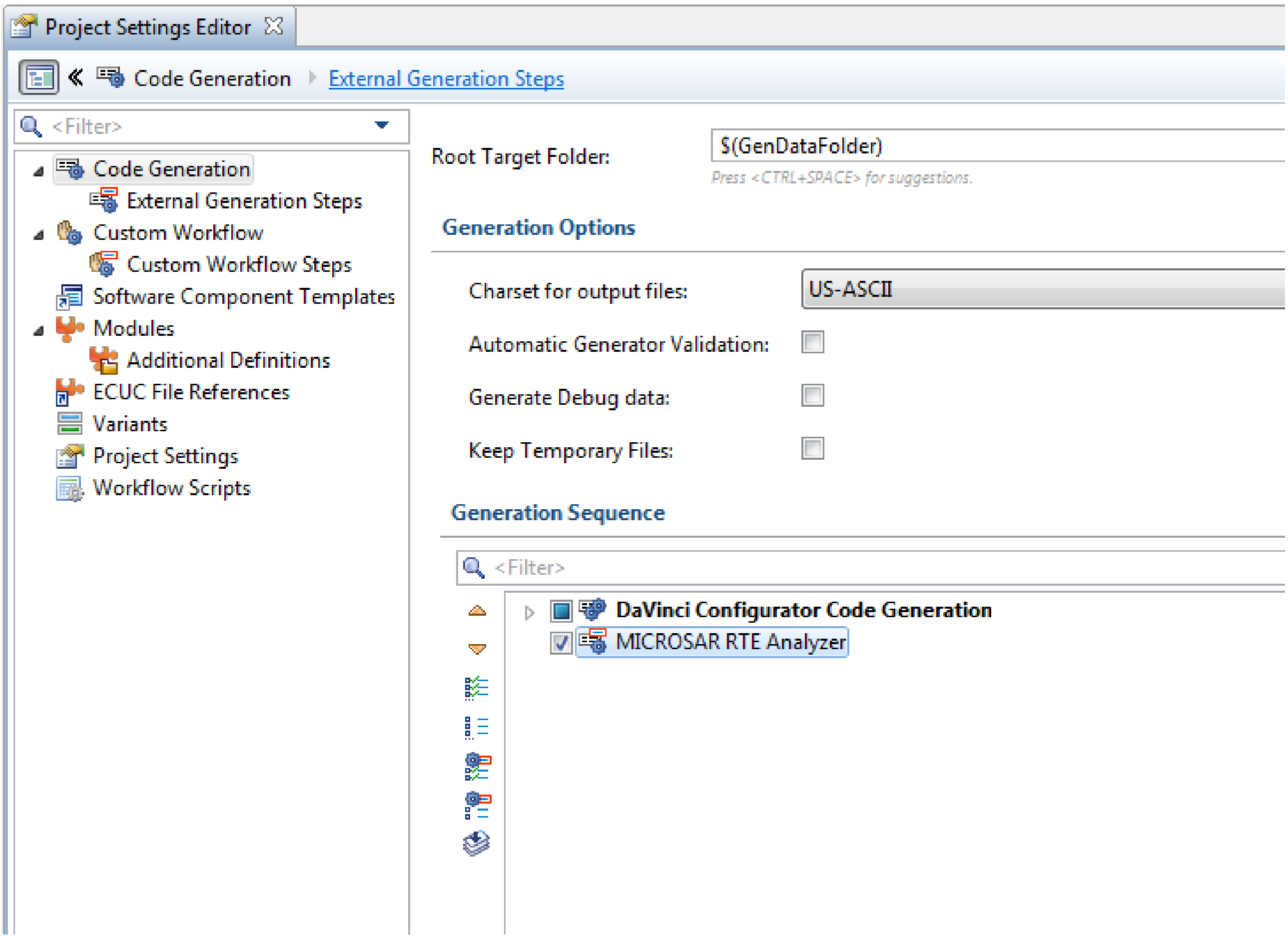
and command line arguments e.g.

`-c $(GenDataFolder)/RteAnalyzer/RteAnalyzerConfiguration.json -o $(GenDataFolder)/RteAnalyzer/Reports`

For Virtual Target, $(GenDataVTTFolder) needs to be used.

Note
It is required to set a working directory for a post generation step.

Now the external generation step needs to be configured to be run after the DaVinci Generators. To configure this click on the item “Code Generation”.



Now select the MICROSAR RTE Analyzer Generation Step and enable it by checking the check box in front of it. Additionally MICROSAR RTE Analyzer should be run after DaVinci Configurator Pro generated the data. Therefore it is necessary to move it after the DaVinci Code Generation using the Down button with the “ ▾ ” symbol.

Now MICROSAR RTE Analyzer will be automatically executed after the DaVinci Configurator Pro has generated the data.

Note
MICROSAR RTE Analyzer will also be executed if the data was not successfully generated.

5 Glossary and Abbreviations

5.1 Glossary

Term	Description
DaVinci CFG	DaVinci Configurator 5: The BSW and RTE Configuration Editor.

Table 5-1 Glossary

5.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
EAD	Embedded Architecture Designer
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
PPORT	Provide Port
RPORT	Require Port
RTE	Runtime Environment
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 5-2 Abbreviations

6 Additional Copyrights

MICROSAR RTE Analyzer contains Free and Open Source Software (FOSS). The following table lists the files which contain this software, the kind and version of the FOSS, the license under which this FOSS is distributed and a reference to a license file which contains the original text of the license terms and conditions. The referenced license files can be found in the directory of MICROSAR RTE Analyzer.

File	FOSS	License	License Reference
MicrosarRteAnalyzer.exe MicrosarRteAnalyzerCfgGen.exe	Perl 5.20.2	Artistic License	License_Artistic.txt
MicrosarIRAnalyzer.exe llvm-link.exe	llvm 3.6.2 vssa r343	LLVM License	License_LLVM.txt
clang.exe	Clang 3.6.2	LLVM License	License_LLVM.txt

Table 6-1 Free and Open Source Software Licenses

7 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com