Components and Tools for Functional Safety Applications

May, 2021

White Paper

Introduction

Functional safety is important and prevalent across a variety of markets, including the automotive, industrial, medical, and railway sectors, and sometimes even in consumer electronics. A failure of a safety-critical system may cause high costs or even endanger human beings. With the unbroken trend toward growing software size in embedded systems, more and more safety-critical functionality is being implemented. Furthermore, due to increasing connectivity requirements, including cloud-based services, device-to-device communication, and over- the-air updates, more and more security issues are arising in safety-critical software as well. Preventing software-induced system failures becomes increasingly important.

There are three important steps that help to tackle this task:

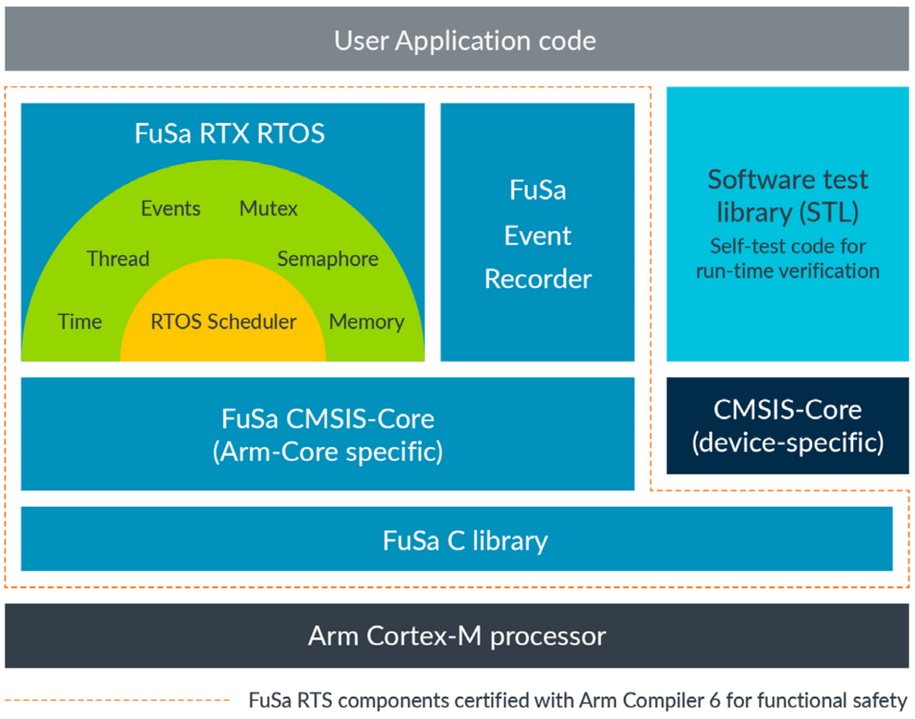
1. Use software building blocks that have been qualified for use in functional safety applications.
2. Use a qualified or a formally verified compiler.
3. Adhere to the strict coding guidelines that are mandated for by various safety standards.

Pre-Qualified Software Components

Creating, developing, and optimizing complex safety-related applications from scratch is challenging, but pre-qualified software components help simplify embedded system development by reducing the time and effort required for the final certification. Arm has bundled a set of certified software components for speeding up final safety certification in a wide range of embedded applications:

Arm FuSa RTS: Our functional safety run-time system enables developers to use the highest safety integrity levels (SIL) for their applications. This set of qualified components is highly optimized for Arm Cortex-M processors. It contains a robust real-time operating system (RTOS), an independent processor abstraction layer, and a verified C library. FuSa RTS is certified by TÜV SÜD for use in a wide range of safety standard certification processes.

Fig. 1
Components included in the FuSa RTS package


User Application code
FuSa RTX RTOS
Events
Mutex
Thread
Semaphore
Time
RTOS Scheduler
Memory
FuSa Event Recorder
Software test library (STL)
Self-test code for run-time verification
FuSa CMSIS-Core (Arm-Core specific)
CMSIS-Core (device-specific)
FuSa C library
Arm Cortex-M processor
FuSa RTS components certified with Arm Compiler 6 for functional safety

Process isolation: The latest version of FuSa RTS supports process isolation to ensure that the non-safety part (or the part with lower integrity level) does not impact the operation of the safety critical part (or the part with higher integrity level) of an application.

The process isolation in FuSa RTS is much more than a simple MPU protection scheme available with other safety qualified RTOS. The spatial isolation using an **MPU protection** scheme is accompanied by **safety classes** that enable access control to RTOS objects allocated with the RTX kernel. All RTOS objects, including threads, are assigned to a safety class value. The threads that belong to a lower safety class cannot modify RTOS objects of a higher safety class.

Furthermore, a temporal isolation of the RTOS threads is accomplished by so-called **thread watchdogs.** In an RTOS, all threads share the computing time of the processor. This sharing must be controlled to avoid any undesired impacts on the execution of safety-critical functions. In FuSa RTS, each thread can maintain its own **thread watchdog**. In case of timing violations, a **thread watchdog alert** is raised.

Finally, a safety critical system must not fail but should always return to a known good state. In case of a failure, for example a HardFault, a memory access violation, or a thread watchdog alert, FuSa RTS blocks the execution of uncritical parts to proceed to a safety state. The RTOS kernel can suspend thread execution, for example for threads in a lower safety class. This can be used to recover execution of the critical thread operation.

Pre-Qualified or Formally Verified Toolchain

The safety certification of products requires a compiler toolchain used in development to be qualified according to appropriate functional safety standards. The process of qualifying these tools known as 'tool qualification' or 'tool validation' can be a time-consuming and expensive process. Moreover, it does not offer any differentiation to the final product. While users are responsible for the overall tool qualification process, vendors of development tools can make this process much easier by offering tools that are qualified to appropriate safety standards. An alternative for tool providers is to offer comprehensive automatic qualification support kits, as available, e.g., for the AbsInt RuleChecker.

Arm Compiler for Functional Safety is a qualified C/C++ toolchain that has been assessed by safety-accredited certification body, TÜV SÜD. The qualified toolchain is suitable for developing embedded software for safety markets including automotive, industrial, medical, railways, and aviation.

With a TÜV certificate and a comprehensive qualification kit, Arm Compiler for Functional Safety greatly simplifies the overall tool qualification process allowing users to focus on their product development.

An alternative is CompCert, an optimizing C compiler intended to compile safety-critical and mission-critical software written in C and meeting high levels of assurance. CompCert is the only production compiler that is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues. The code it produces is proven to behave exactly as specified by the semantics of the source C program.
This level of confidence in the correctness of the compilation process is unprecedented and contributes to meeting the highest levels of software assurance.

Obeying Coding Guidelines

Coding guidelines aim at improving code quality and can be considered a prerequisite for developing safety- or security-relevant software. Obeying coding guidelines is strongly recommended by all current safety standards, including DO-178C, IEC 61508, ISO 26262, and EN 50128. These norms do not enforce compliance to a particular coding guideline but define properties to be checked by the coding standards applied. As an example, the ISO 26262 gives a list of topics to be covered, including enforcement of low complexity, enforcing usage of a language subset, enforcing strong typing, and use of well-trusted design principles (cf. ISO 26262:6, Table 1). The language subset to be enforced should exclude ambiguously defined language constructs, language constructs that could result in unhandled runtime errors, and language constructs known to be error prone.

The MISRA C standard has originally been developed with a focus on automotive industry but is now widely recognized as the predominant coding guideline for safety-critical systems in general. Its goal is to avoid programming errors and enforce a programming style that enables the safest possible use of C. A particular focus is on dealing with undefined/unspecified behavior of C and on preventing runtime errors.

Automatic static analysis tools have gained popularity in software development as they offer a tremendous increase in productivity by automatically checking the code under a wide range of criteria. This includes checking coding guidelines, computing code metrics, and finding runtime errors.

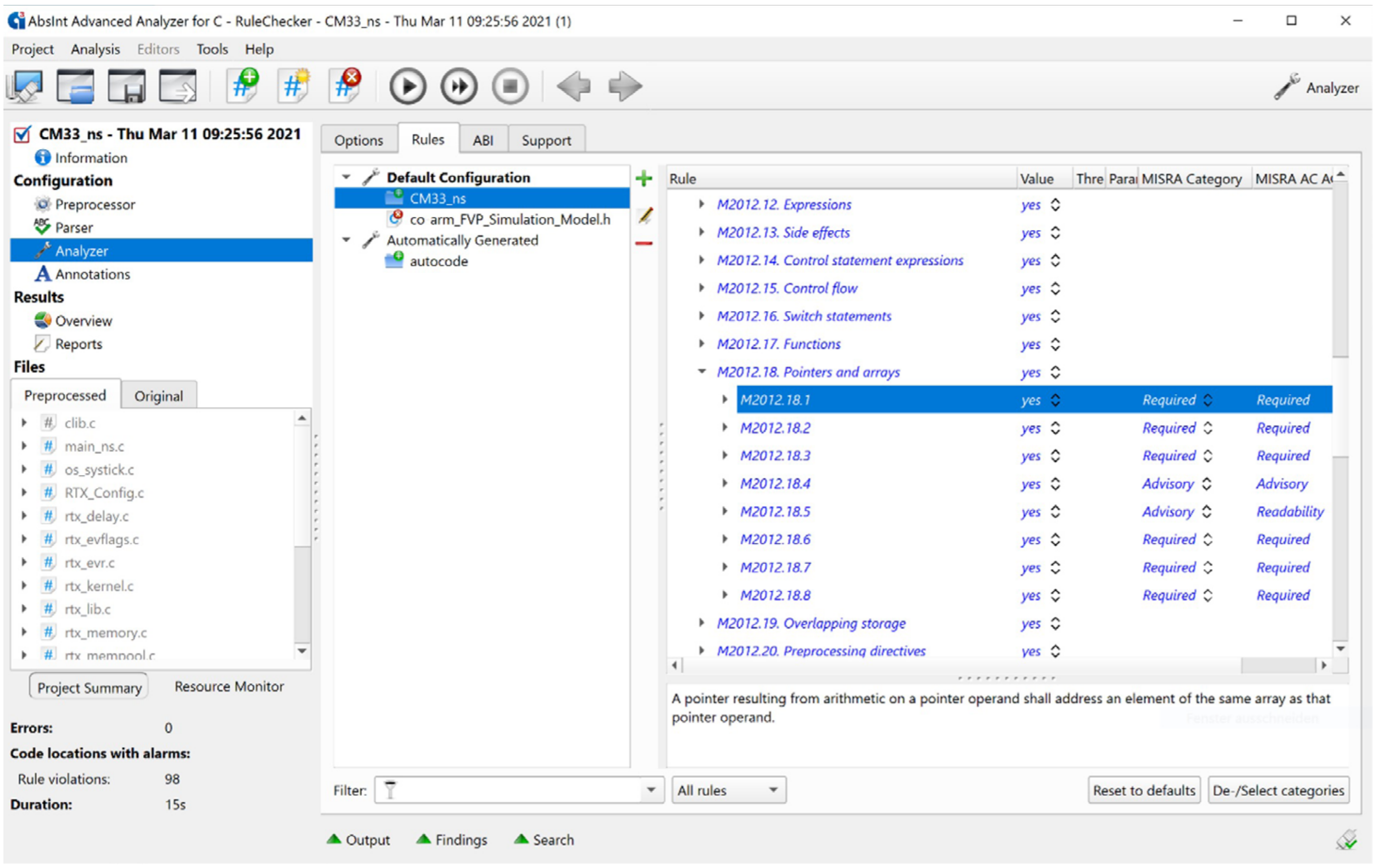
Purely syntactical methods can be applied to check syntactical coding rules as contained in all relevant coding guidelines. Semantical (undecidable) rules require a deeper understanding of the code as they focus on semantical properties which requires knowledge about variable values, pointer targets etc. Sound semantical analyses can provide assurance that certain types of defects do not occur in the code.

The AbsInt RuleChecker is a static analyzer designed to check coding guidelines and compute code metrics for C/C++ programs. It is fast and easy to use. Since 2021, a dedicated plugin enables the seamless integration of RuleChecker in the Keil µVision IDE.

Supported coding guidelines include:

- MISRA C:2004
- MISRA C:2012
- ISO/IEC TS 17961:2013
- SEI CERT Secure C/C++
- MITRE Common Weakness Enumeration (CWE)
- MISRA C++:2008,
- Adaptive AUTOSAR C++14

RuleChecker can be configured in a highly **flexible** way: individual rules and specific aspects of certain rules can be toggled, MISRA guideline recategorization plans are supported, heterogeneous projects with different rule configurations for different software components are supported as well. Rules can also be applied or disapplied at the level of individual files or even code fragments.



clib.c
main_ns.c
os_systick.c
RTX_Config.c
rtx_delay.c
rtx_evflags.c
rtx_evr.c
rtx_kernel.c
rtx_lib.c
rtx_memory.c
rtx_mempool.c

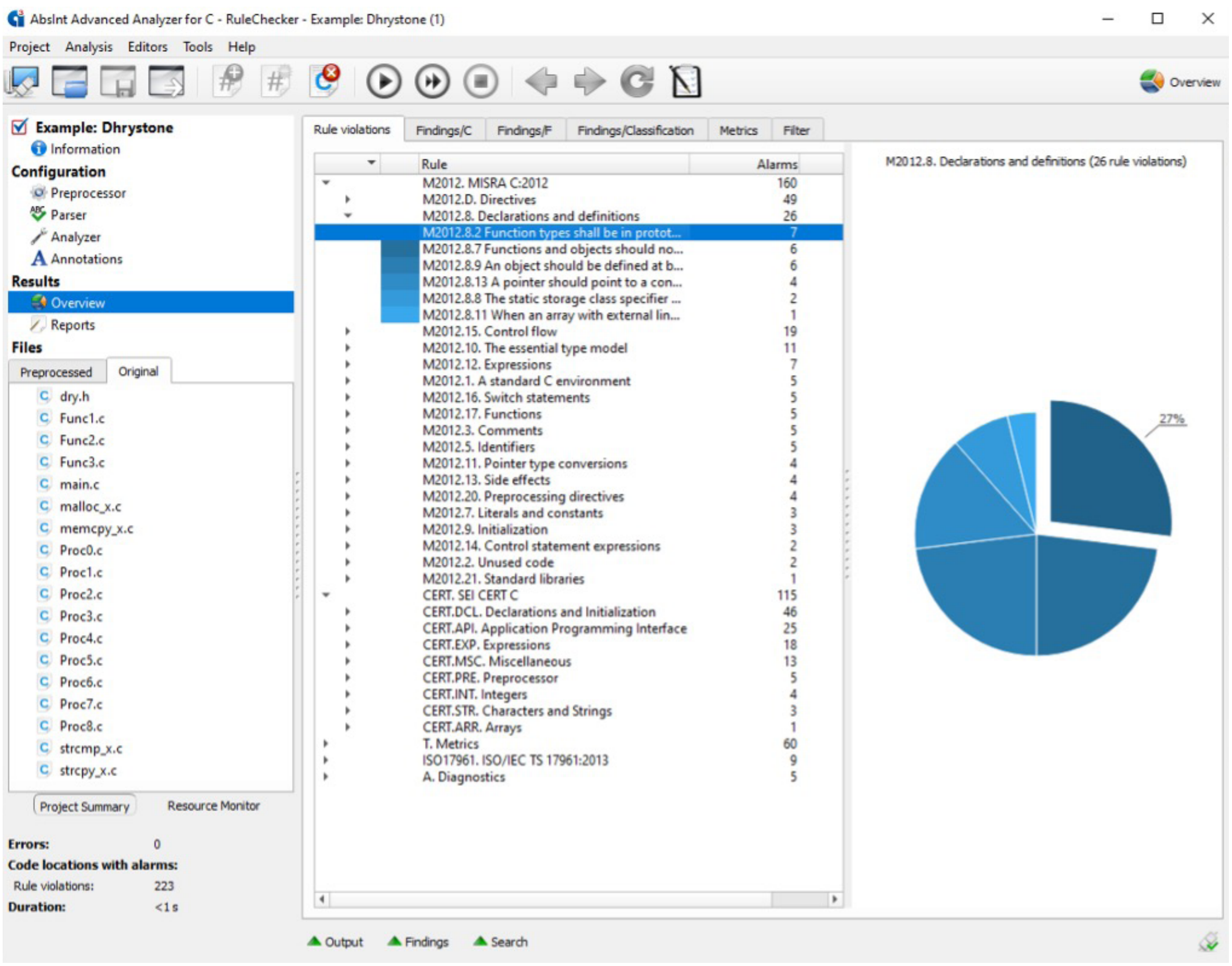
Rule	Value	Thre	Para	MISRA Category	MISRA AC A
M2012.12. Expressions	yes				
M2012.13. Side effects	yes				
M2012.14. Control statement expressions	yes				
M2012.15. Control flow	yes				
M2012.16. Switch statements	yes				
M2012.17. Functions	yes				
M2012.18. Pointers and arrays	yes				
M2012.18.1	yes			Required	Required
M2012.18.2	yes			Required	Required
M2012.18.3	yes			Required	Required
M2012.18.4	yes			Advisory	Advisory
M2012.18.5	yes			Advisory	Readability
M2012.18.6	yes			Required	Required
M2012.18.7	yes			Required	Required
M2012.18.8	yes			Required	Required
M2012.19. Overlapping storage	yes				
M2012.20. Preprocessing directives	yes				

A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

Errors:	0
Code locations with alarms:	
Rule violations:	98
Duration:	15s

Fig. 2
AbsInt RuleChecker rule set configuration

Multiple result views and graphical visualizations enable an **efficient result exploration**. Report files covering all aspects of the results can be generated in open formats, including XML, ASCII-text and html. RuleChecker is **fully batch-mode compatible** and can be used in **continuous integration** frameworks. Further plugins to other third-party tools are available.

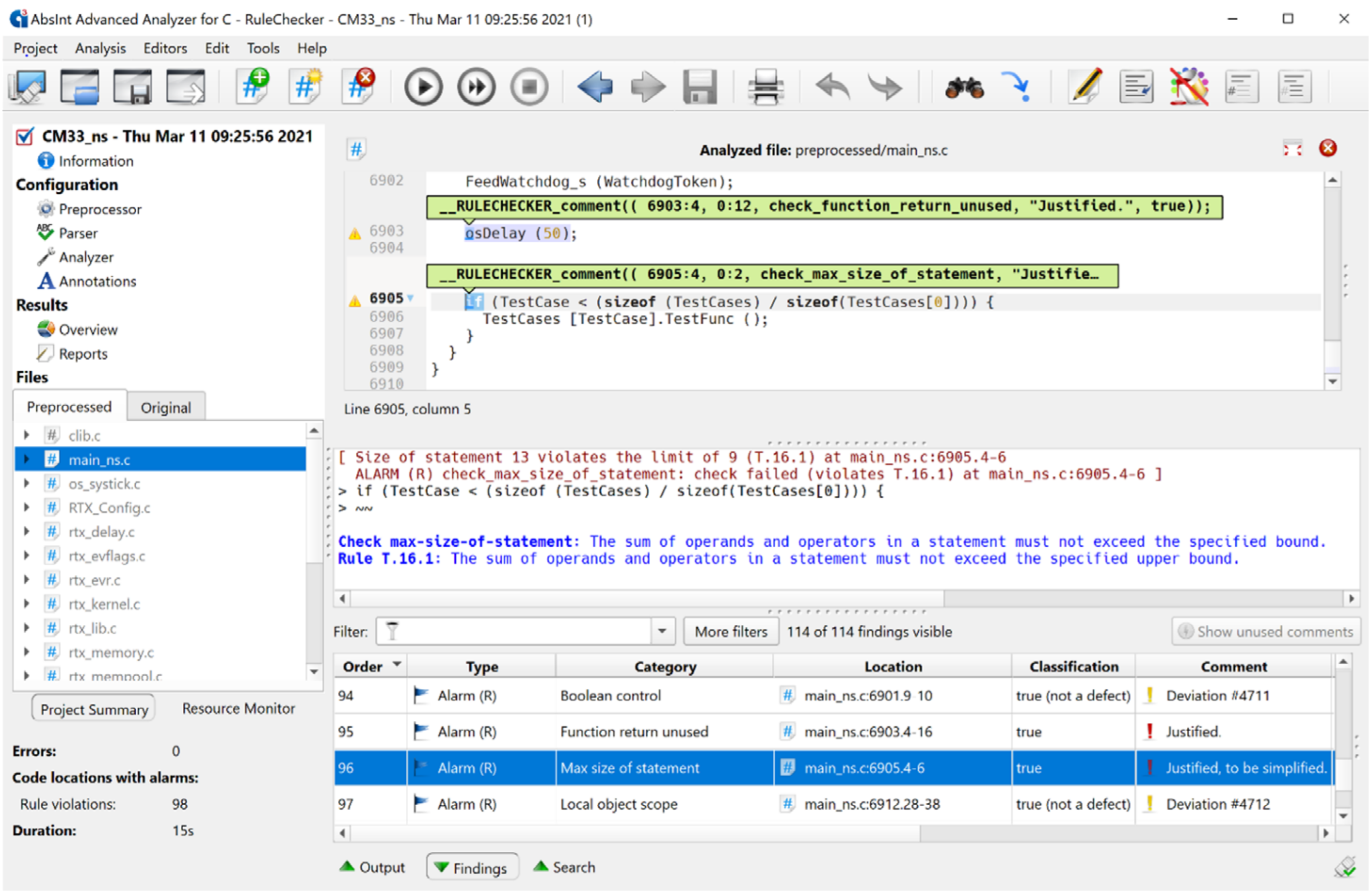


Rule	Alarms
M2012. MISRA C:2012	160
M2012.D. Directives	49
M2012.8. Declarations and definitions	26
M2012.8.2 Function types shall be in protot...	7
M2012.8.7 Functions and objects should no...	6
M2012.8.9 An object should be defined at b...	6
M2012.8.13 A pointer should point to a con...	4
M2012.8.8 The static storage class specifier ...	2
M2012.8.11 When an array with external lin...	1
M2012.15. Control flow	19
M2012.10. The essential type model	11
M2012.12. Expressions	7
M2012.1. A standard C environment	5
M2012.16. Switch statements	5
M2012.17. Functions	5
M2012.3. Comments	5
M2012.5. Identifiers	5
M2012.11. Pointer type conversions	4
M2012.13. Side effects	4
M2012.20. Preprocessing directives	4
M2012.7. Literals and constants	3
M2012.9. Initialization	3
M2012.14. Control statement expressions	2
M2012.2. Unused code	2
M2012.21. Standard libraries	1
CERT. SEI CERT C	115
CERT.DCL. Declarations and Initialization	46
CERT.API. Application Programming Interface	25
CERT.EXP. Expressions	18
CERT.MSC. Miscellaneous	13
CERT.PRE. Preprocessor	5
CERT.INT. Integers	4
CERT.STR. Characters and Strings	3
CERT.ARR. Arrays	1
T. Metrics	60
ISO17961. ISO/IEC TS 17961:2013	9
A. Diagnostics	5

Errors:	0
Code locations with alarms:	
Rule violations:	223
Duration:	<1 s

Fig. 3
Overview of rule violations
in AbsInt RuleChecker

Rule violations can be **classified** and **commented**, either externally to the code in a robust line-independent way, or by automatically created source code comments. Identifying and tracking new rule violations triggered by source code changes is easily possible.



Analyzed file: preprocessed/main_ns.c

```
6902      FeedWatchdog_s (WatchdogToken);
        __RULECHECKER_comment(( 6903:4, 0:12, check_function_return_unused, "Justified.", true));
6903      osDelay (50);
6904

        __RULECHECKER_comment(( 6905:4, 0:2, check_max_size_of_statement, "Justifie…
6905      if (TestCase < (sizeof (TestCases) / sizeof(TestCases[0]))) {
6906        TestCases [TestCase].TestFunc ();
6907      }
6908    }
6909  }
6910
```

Line 6905, column 5

```
[ Size of statement 13 violates the limit of 9 (T.16.1) at main_ns.c:6905.4-6
  ALARM (R) check_max_size_of_statement: check failed (violates T.16.1) at main_ns.c:6905.4-6 ]
> if (TestCase < (sizeof (TestCases) / sizeof(TestCases[0]))) {
> ~~

Check max-size-of-statement: The sum of operands and operators in a statement must not exceed the specified bound.
Rule T.16.1: The sum of operands and operators in a statement must not exceed the specified upper bound.
```

Filter: 114 of 114 findings visible

Order	Type	Category	Location	Classification	Comment
94	Alarm (R)	Boolean control	main_ns.c:6901.9 10	true (not a defect)	Deviation #4711
95	Alarm (R)	Function return unused	main_ns.c:6903.4-16	true	Justified.
96	Alarm (R)	Max size of statement	main_ns.c:6905.4-6	true	Justified, to be simplified.
97	Alarm (R)	Local object scope	main_ns.c:6912.28-38	true (not a defect)	Deviation #4712

Errors:	0
Code locations with alarms:	
Rule violations:	98
Duration:	15s

Fig. 4
Classification and justification of findings in AbsInt RuleChecker

With the µVision plugin, the RuleChecker configuration is automatically derived from the project configuration **without need for manual user interaction**. Compiler-specific types and macros are automatically exported and made available to RuleChecker. A hierarchical project configuration allows users to include configuration templates and create their own configuration defaults.

RuleChecker can be coupled with the sound static analyzer **Astrée** that finds runtime errors, such as buffer overflows and data races, and can prove their absence. It can be **automatically qualified** according to all relevant safety norms, including ISO 26262, DO-178B/C, IEC 61508, EN 50128, up to the highest criticality levels.

Conclusion

Arm offers software components and toolchains for functional safety applications. Together with third-party tooling from AbsInt, users can certify their functionals safety applications in less time. For further information, watch this video or directly try Keil MDK and AbsInt RuleChecker.