

MathWorks
AUTOMOTIVE
CONFERENCE
2018

정형 기법을 활용한 AUTOSAR
SWC의 구현 확인 및 정적 분석

Develop high quality embedded software

이 영준
Principal Application Engineer



Agendas

- **Unit-proving of AUTOSAR Component and Runtime error**
- **Secure Coding Standard and Polyspace**
 - MISRA-C:2012 Amendment 1
 - ISO 17961
 - CERT-C/C++
 - CWE

Unit-Proving of AUTOSAR Component And Runtime Error

What is AUTOSAR?

- The Automotive industry and its challenges



OEMs objectives:

- Integration from different suppliers
- Need confidence in the supplier's code

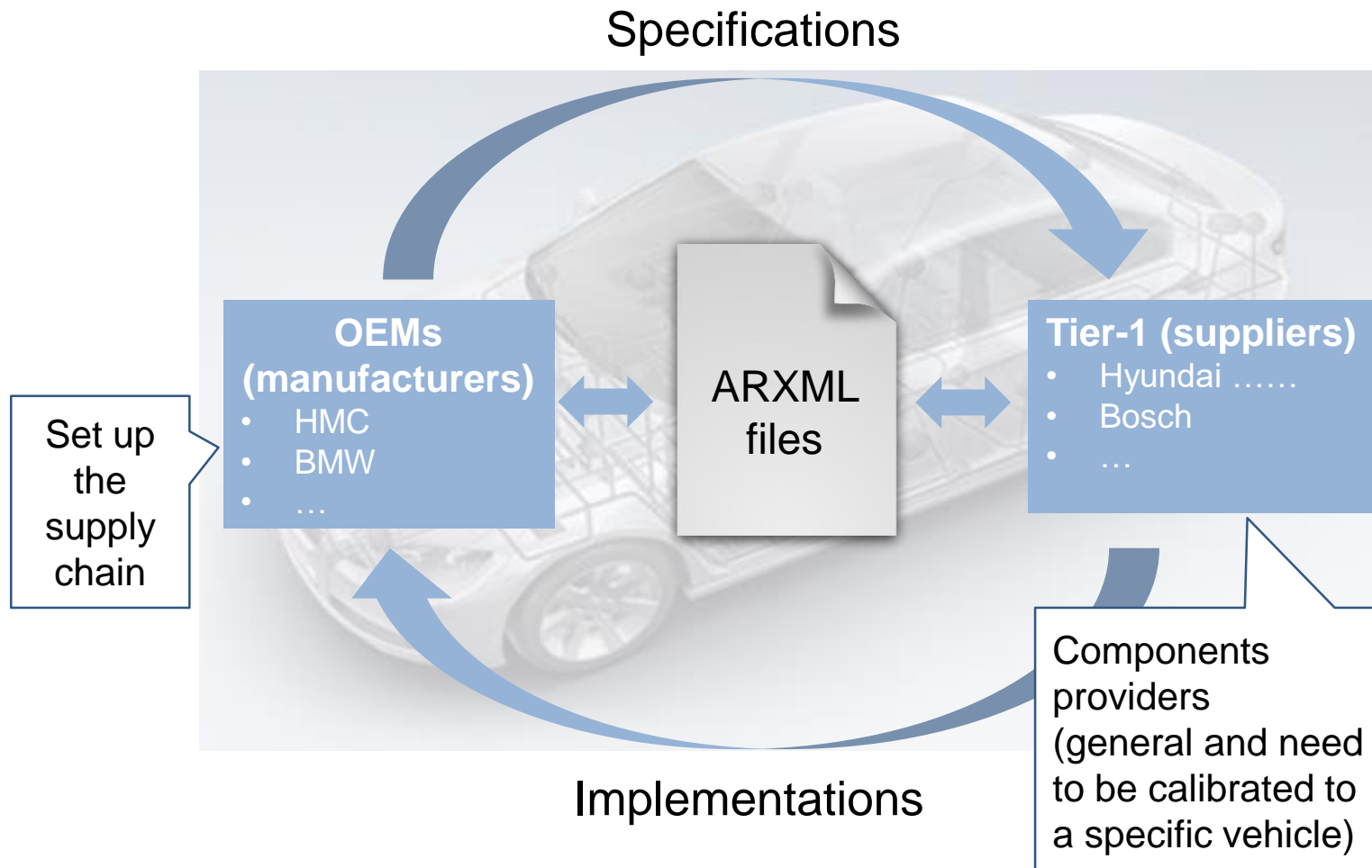
Supplier's challenge:

- Time-to-market
- Code size
- Pressure from OEMs

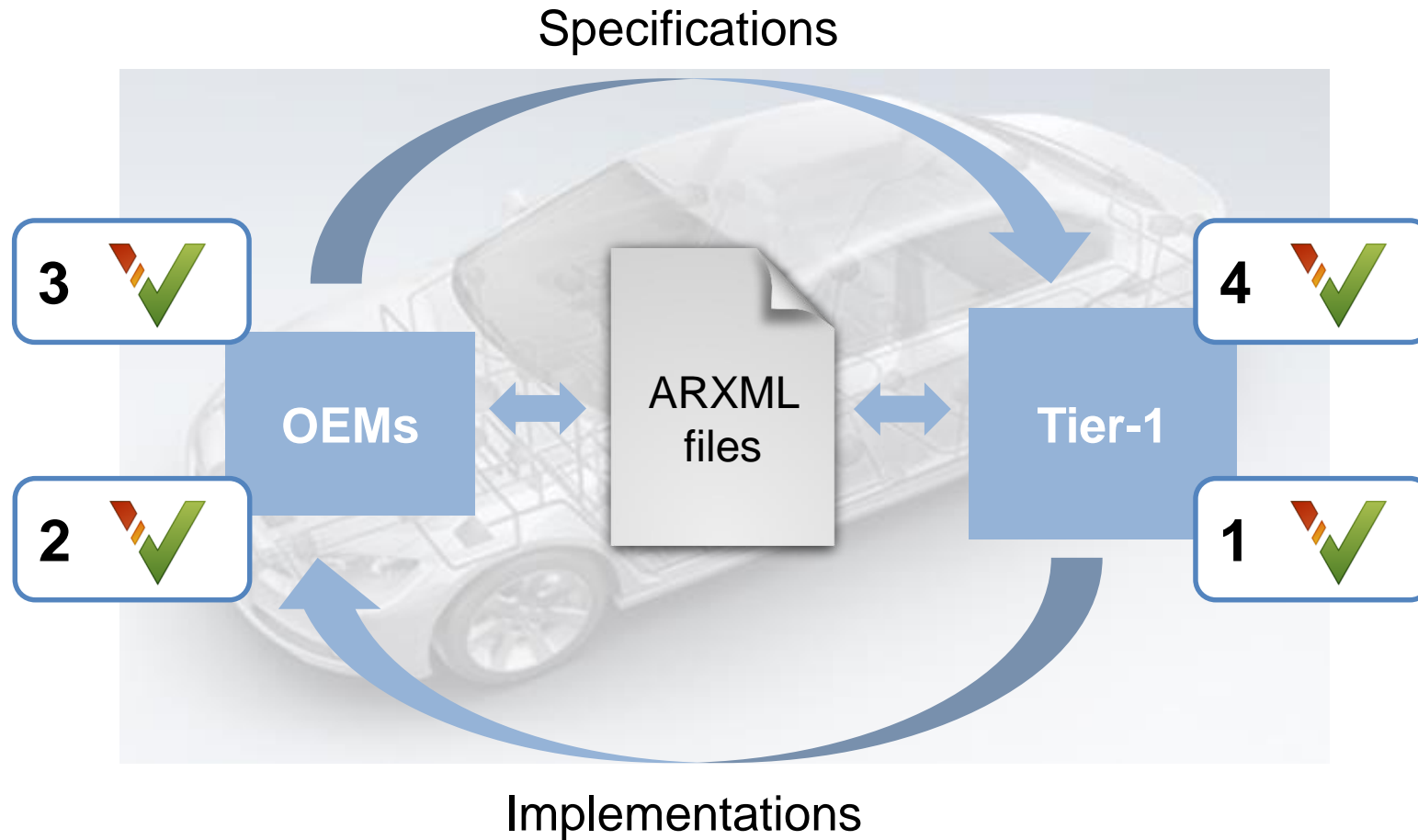
AUTOSAR solves by providing a software architecture and common specifications (ARXML files)



Need for validation of AUTOSAR components among actors



How Polyspace for AUTOSAR can help?

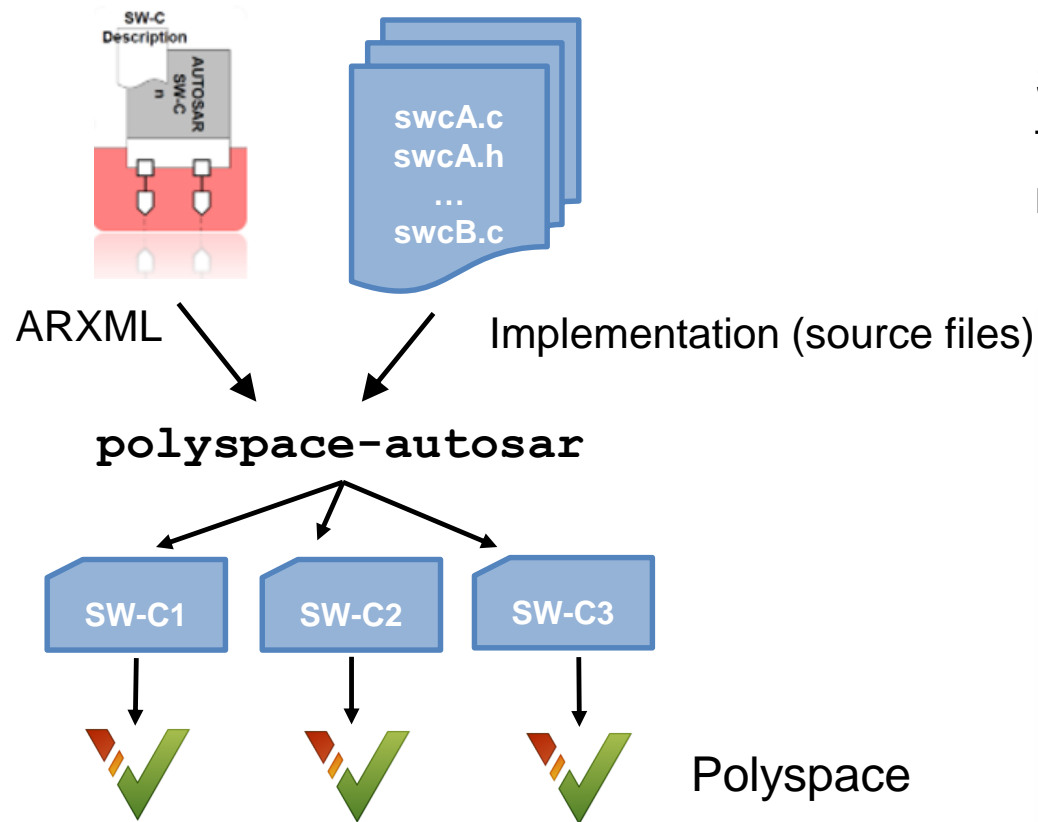


- 1) check for run-time errors and mismatch in the ARXML specifications
- 2) Check if implementation follow specifications
- 3) assess impact of changes in the specifications
- 4) check implementation against specifications updates

ARXML files are used to communicate, Polyspace for AUTOSAR is used to prove robustness and compliance

Polyspace for AUTOSAR features

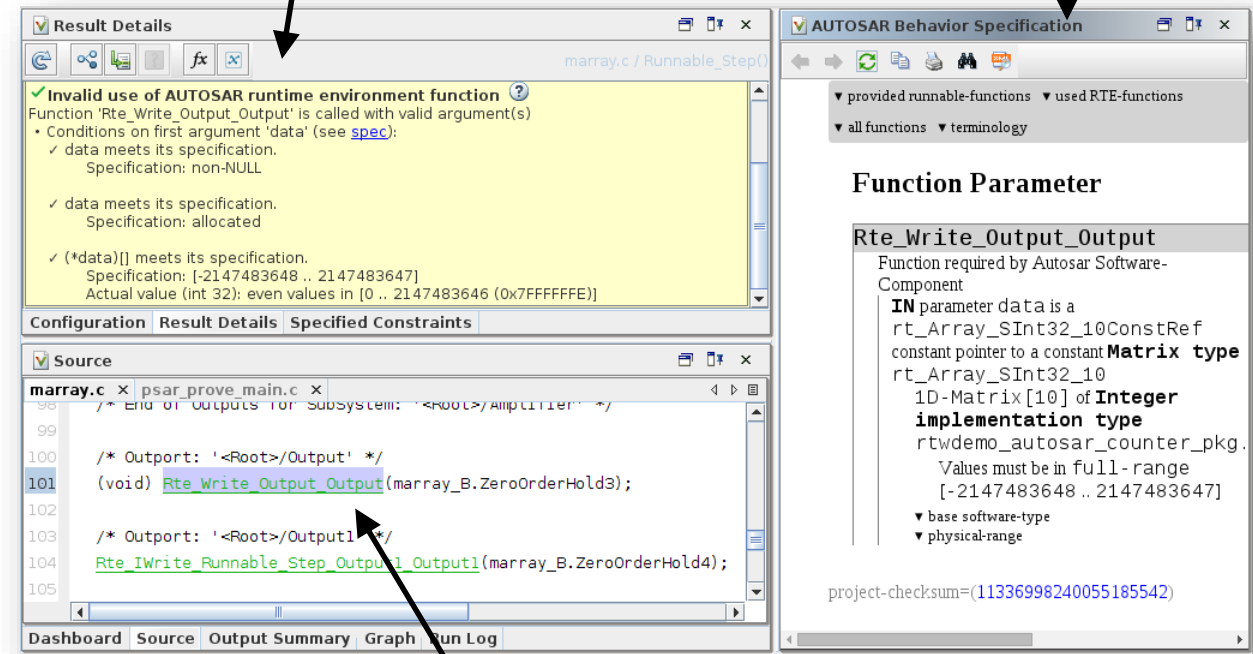
Automatic split by component



Automatic launching on each component

Sound analysis

Sound analysis plus checks to prove that the code matches the specification



Prove specs matching

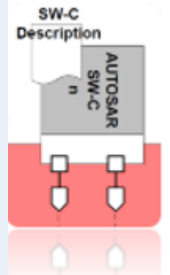
Back to specifications

New view to detail the AUTOSAR specification

Polyspace for AUTOSAR workflow

Specifications

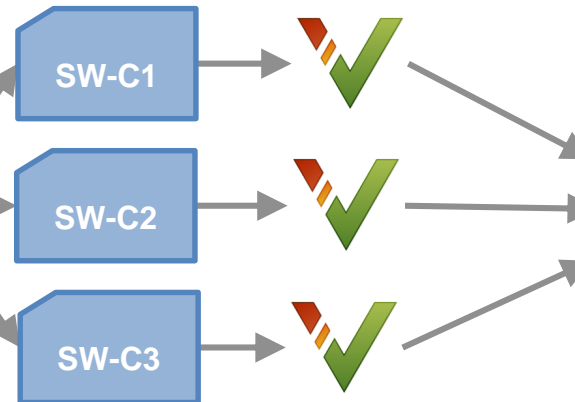
ARXML files



Split the code in components as defined in ARXML

Polyspace for AUTOSAR

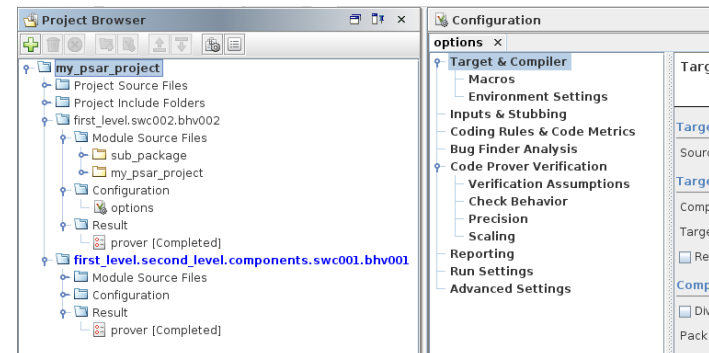
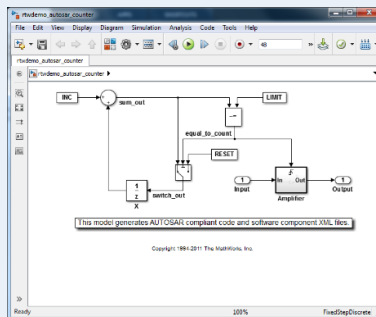
Perform a separate unit analysis of each component with Polyspace



- ✓ Free of run-time errors
- ✓ Checks that code of runnable respects its output specification
- ✓ Checks that code of runnable calls Rte functions in respect of their specification

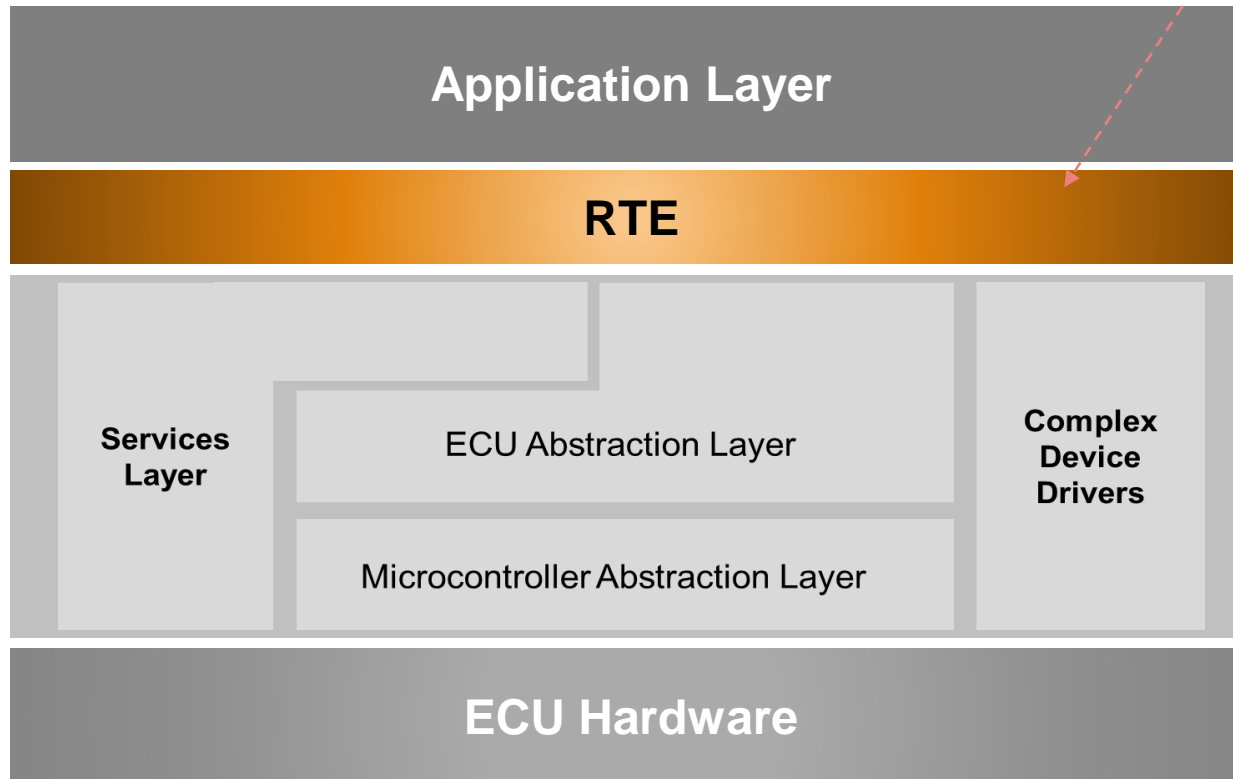
Implementation

Simulink model

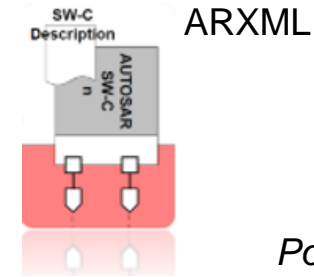


Polyspace and AUTOSAR

AUTOSAR architecture



*ARXML provides
specification
of Application Layer
and link with RTE*



*Polyspace verifies
the match between
code and ARXML*

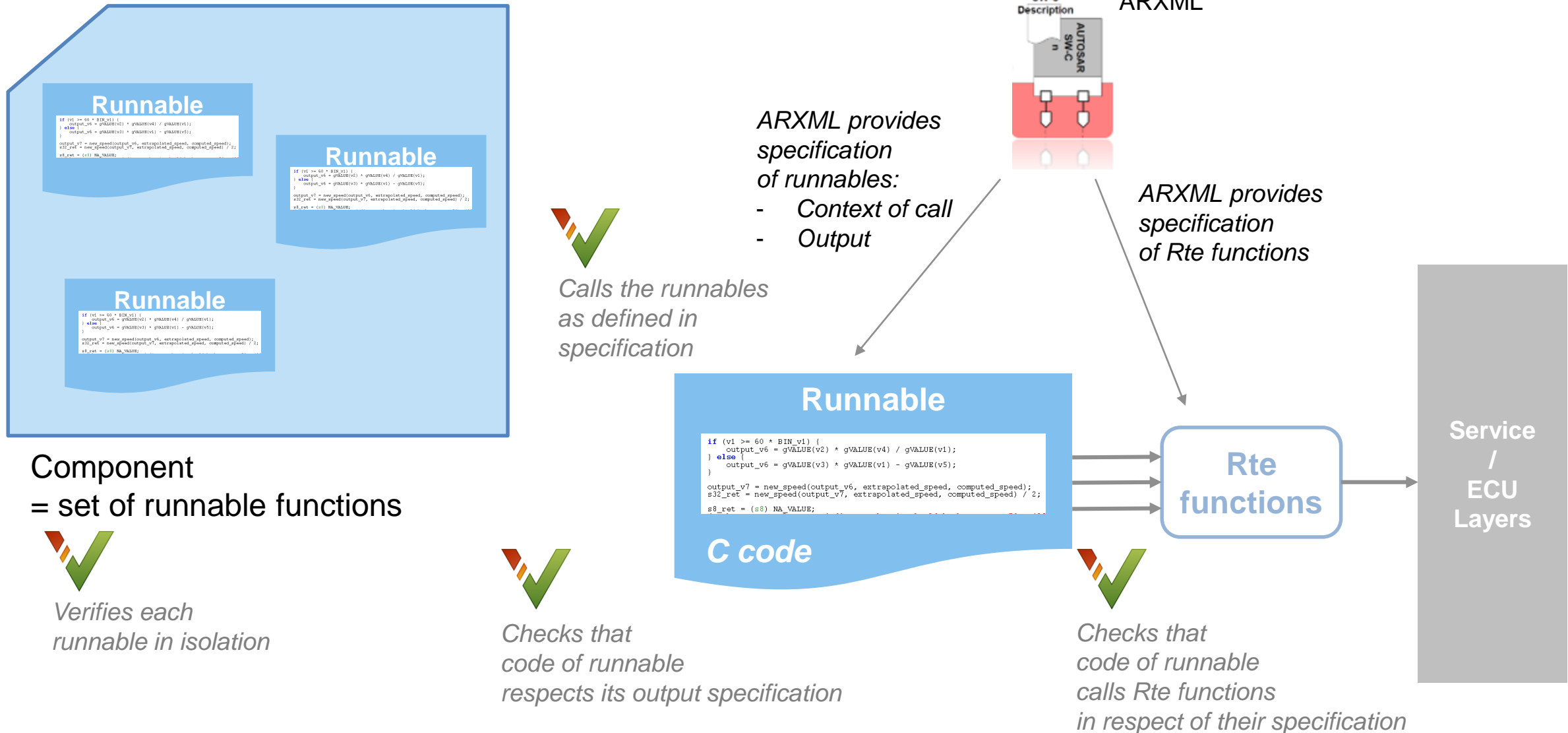


Polyspace verifies the Application Layer

*Polyspace stubs the RTE Layer
RTE Layer not verified by "Polyspace for AUTOSAR"
Polyspace can verify RTE*

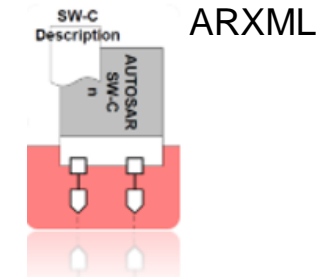
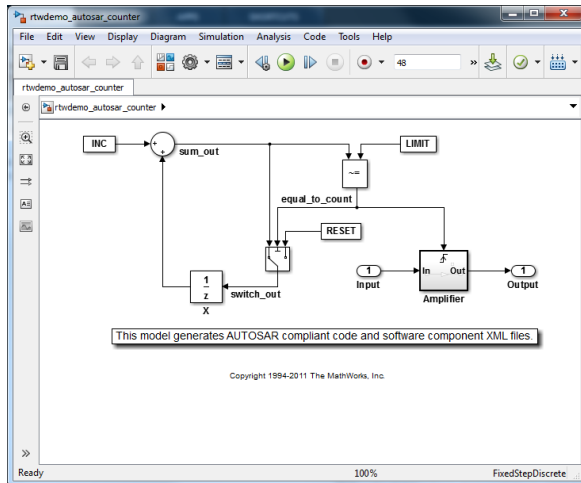
*Not verified by "Polyspace for AUTOSAR"
Polyspace may verify these*

Unit verification of an AUTOSAR software component



Unit verification of an AUTOSAR software component

Simulink model



ARXML provides
specification
of runnables:

- Context of call
- Output

ARXML provides
specification
of Rte functions



*Calls the runnables
as defined in
specification*

Runnable

```
if (v1 >= 60 * BIN_v1) {
    output_v6 = gVALUE(v2) * gVALUE(v4) / gVALUE(v1);
} else {
    output_v6 = gVALUE(v3) * gVALUE(v1) - gVALUE(v5);
}

output_v7 = new_speed(output_v6, extrapolated_speed, computed_speed);
s32_reE = new_speed(output_v7, extrapolated_speed, computed_speed) / 2;
s8_ret = (s8) NA_VALUE;
```

C code



*Checks that
code of runnable
respects its output specification*

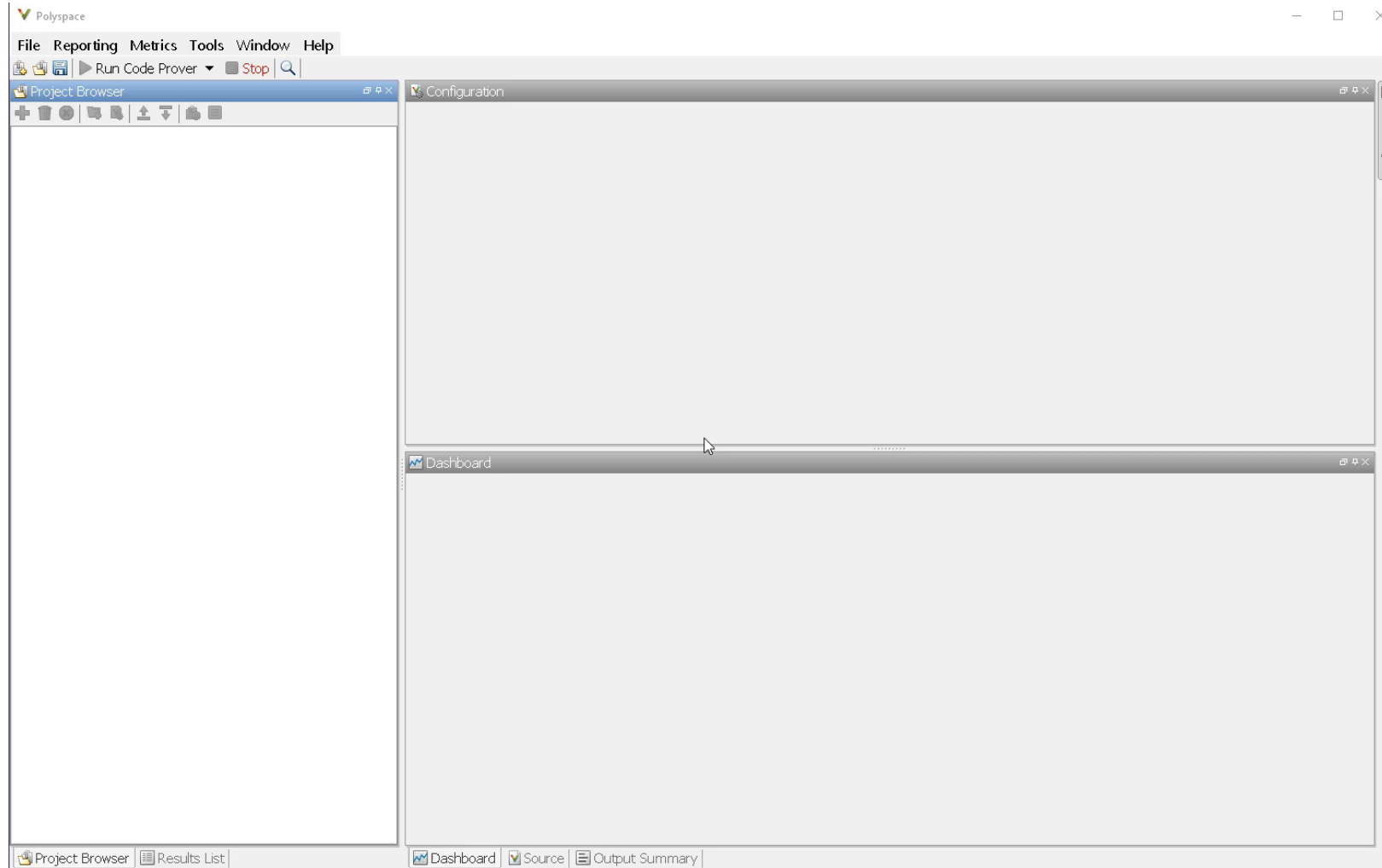
Rte
functions

*Checks that
code of runnable
calls Rte functions
in respect of their specification*

Service
/
ECU
Layers

Hand-Written Code based on ARXML

Polyspace for AUTOSAR SWC



Generated Code From Simulink Model based on ARXML

Polyspace for AUTOSAR SWC

The screenshot displays the Simulink Embedded Coder environment for a model named 'aswc_sbr'. The main workspace shows a block diagram titled 'Polyspace for AUTOSAR Demo with Seat Belt Reminder Design Model'. The diagram includes three input ports on the left: 'KEY' (labeled 1), 'SeatBeltFasten' (labeled 2), and 'Speed' (labeled 3). These inputs feed into a central block labeled 'SBR_WrongOutput'. Inside this block, there are two sub-blocks: 'SeatBeltFasten' and 'SeatBeltIcon', which are interconnected with feedback loops. The output of the 'SeatBeltIcon' block is connected to an output port on the right labeled 'SeatBeltIcon' (labeled 1). The interface also features a left sidebar with configuration options, a right sidebar with a Property Inspector, and a bottom section with a Model Data Editor table.

Model Data Editor

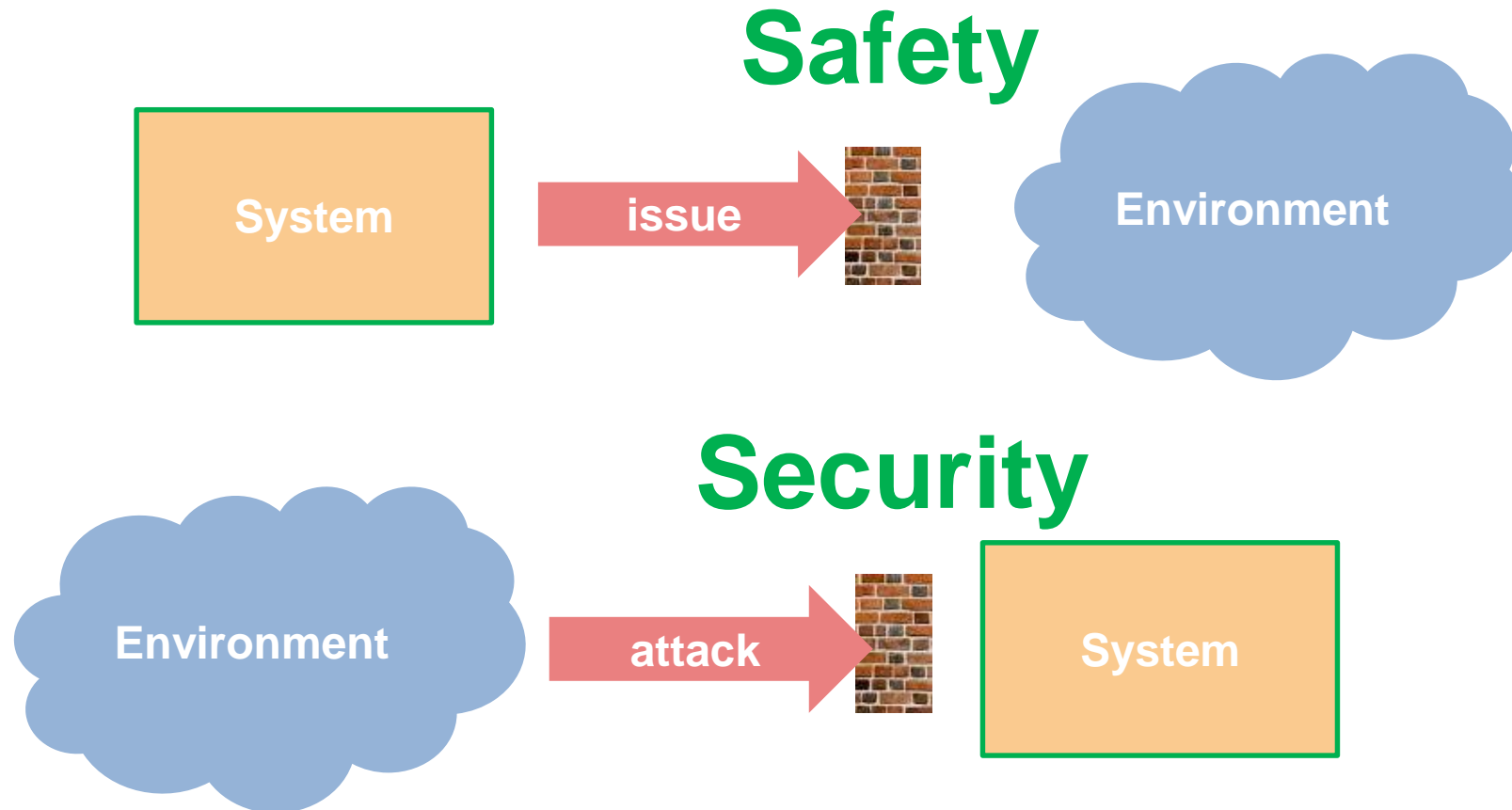
Source	#	Signal Name	Data Type	Min	Max	Dimensions	Complexity	Sample Time	Unit	Resolve
KEY	1		int8	int8 0	0 2	2 -1	1 auto	real -1 [0.001 0]	inherit	<input type="checkbox"/>
SeatBeltFasten	2		int8	int8 0	0 1	1 -1	1 auto	real -1 [0.001 0]	inherit	<input type="checkbox"/>
Speed	3		single	single -200	-200 350	350 -1	1 auto	real -1 [0.001 0]	inherit	<input type="checkbox"/>
SeatBeltIcon	1		int8	int8 0	0 1	1 -1	1 auto	real -1 [0.001 0]	inherit	<input type="checkbox"/>

Workflow Benefits

- Provide **automatically** the best configuration for Polyspace
- Detect **inconsistencies** between AUTOSAR **specifications** and code **implementation**
- **Unit** verification of AUTOSAR software components with Polyspace
 - ✓ **Sound** analysis: proves that code **respects** the specification
 - ✓ **Static** analysis: considers **all** potential cases

Secure Coding and Polyspace

Safety vs. Security



Note: Security issues may cause safety issues

Cybersecurity – Industry Activities & Standards

SAE – Vehicle Cybersecurity Systems Engineering Committee

- [SAE J3061](#) - Cybersecurity Guidebook for Cyber-Physical Vehicle Systems
- [SAE J3101](#) - Requirements for Hardware-Protected Security for Ground Vehicle Applications (WIP)
- [SAE “Cybersecurity Assurance Testing Task Force” \(TEVEES18A1\)](#)

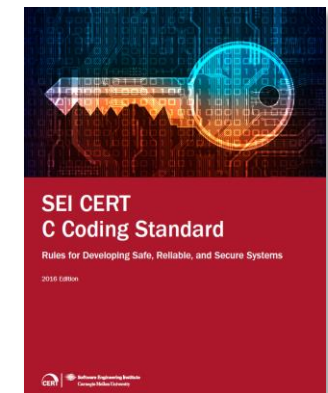
Coding standards & practices that we observe at automotive customers

- [MISRA-C:2012](#) Amendment 1
- [ISO/IEC TS 17961](#) – C Secure Coding Rules
- [CERT-C](#) / [CERT-C++](#)
- [CWE](#) – Common Weakness Enumeration

ISO/IEC TS 17961 Compared with Other Standards

Coding Standard	C Standard	Security Standard	Safety Standard	International Standard	Whole Language
CWE	None/all	Yes	No	No	N/A
MISRA C:2004	C89	No	Yes	No	No
MISRA C:2012	C99	No	Yes	No	No
CERT C99	C99	Yes	No	No	Yes
CERT C11	C11	Yes	Yes	No	Yes
ISO/IEC TS 17961	C11	Yes	No	Yes	Yes

Table is based on the book:



SEI CERT C Coding Standard

- This coding standard consists of ***rules*** and ***recommendations***, collectively referred to as *guidelines*.
- **Rules** are meant to provide normative requirements for code, whereas
- **Recommendations** are meant to provide guidance that, when followed, should improve the safety, reliability, and security of software systems.

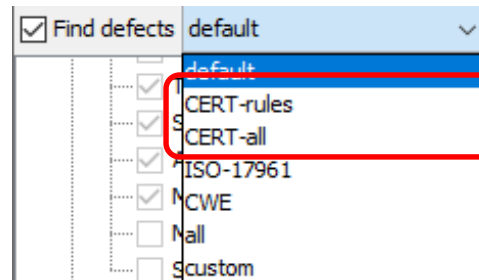
CERT-C Coverage with Polyspace

- You can **map** Polyspace results to CERT C rules and recommendations
- Using Polyspace results, you can address **103** CERT C rules (**90%**) and **95** CERT-C recommendations (**50%**)
 - The [CERT C website](#), under continuous development, lists 118 rules and 188 recommendations (Count based on The CERT C++ Coding Standard document, 2016 Edition)



CERT-C++ coverage with Polyspace

- You can map Polyspace results to CERT C++ rules
- Using the Polyspace results, you can address **34** CERT C++ rules (**40%**) and **79** CERT C rules that also apply to C++ (**99%**)
 - The [CERT C++ website](#), under continuous development, lists 163 rules including 80 CERT C rules that also apply to C++ (based on count in April 2018 in CERT-C++ web site)
- ✓ Two new arguments for option `-checkers` in C++ mode (`-lang CPP`): `CERT-rules` (only CERT-C++ rules) and `CERT-all` (it includes also CERT-C rules that apply)



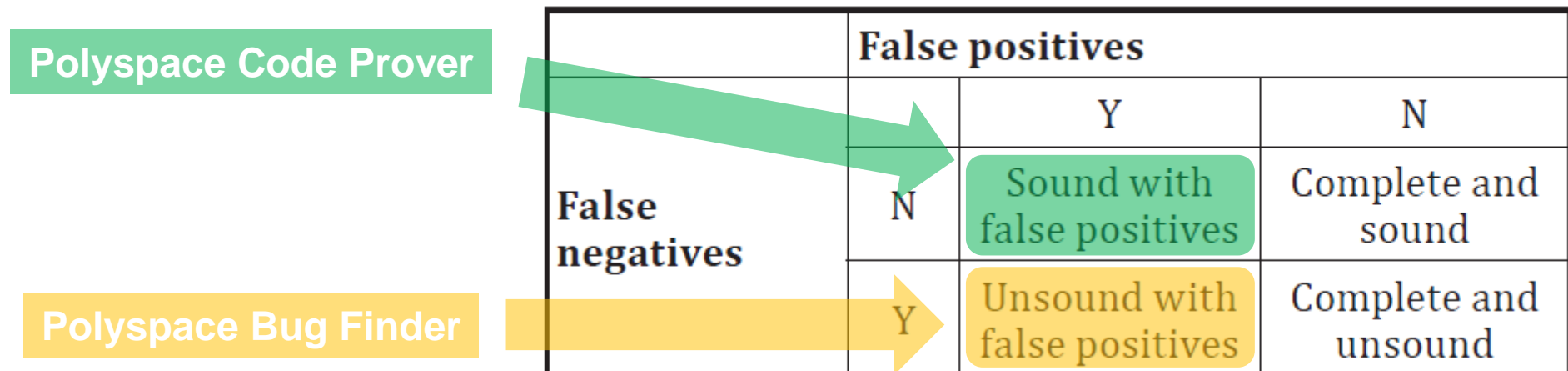
Completeness And Soundness

From ISO 17961

- False Negatives
 - Failure to report a real flaw in the code is usually regarded as the most serious analysis error, as it may leave the user with a false sense of security.
- False Positives
 - The tool reports a flaw when one does not exist.

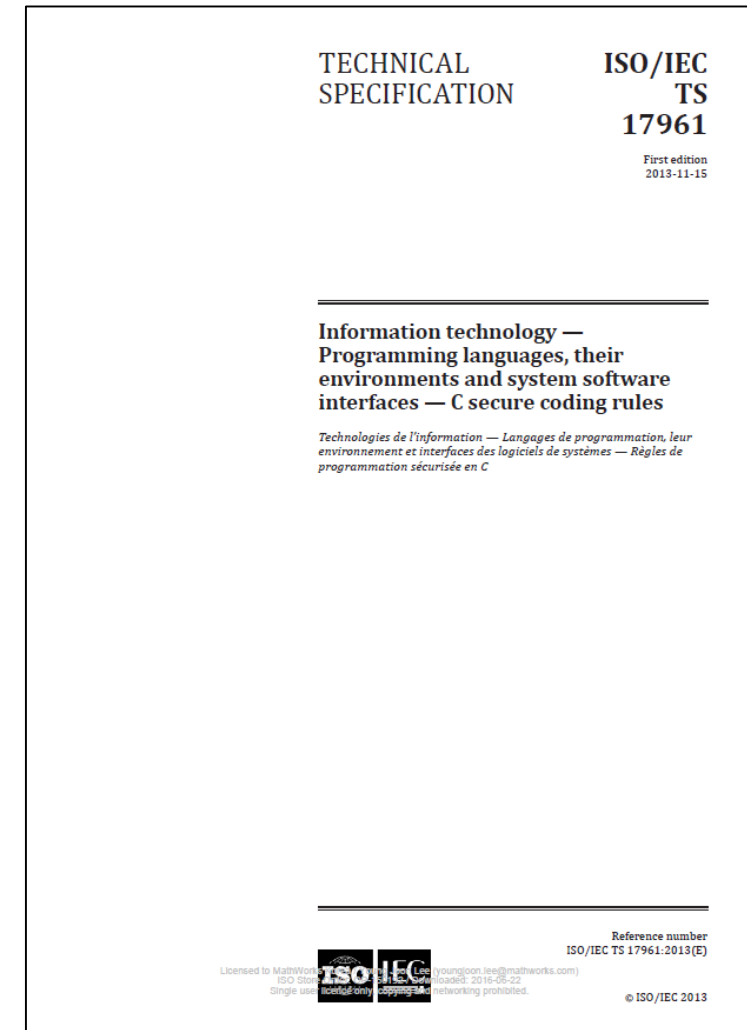
Table 1 — Completeness and soundness

	False positives		
		Y	N
False negatives	N	Sound with false positives	Complete and sound
	Y	Unsound with false positives	Complete and unsound



ISO/IEC TS 17961

C secure coding rules



ISO 17961 - C Secure Coding Rules

- The purpose of this Technical Specification is to specify analyzable secure coding rules that can be automatically enforced to detect security flaws in C-conforming applications.
- To be considered a security flaw, a software bug must be triggerable by the actions of a malicious user or attacker.

ISO 17961 - C Secure Coding Rules

5.1	Accessing an object through a pointer to an incompatible type [ptrcomp].....	5
5.2	Accessing freed memory [accfree].....	6
5.3	Accessing shared objects in signal handlers [accsig].....	7
5.4	No assignment in conditional expressions [boolasgn].....	8
5.5	Calling functions in the C Standard Library other than abort, _Exit, and signal from within a signal handler [asynsig].....	9
5.6	Calling functions with incorrect arguments [argcomp].....	11
5.7	Calling signal	12
5.8	Calling system	13
5.9	Comparison of p	14
5.10	Converting a poi	15
5.11	Converting poin	16
5.12	Copying a FILE	17
5.13	Declaring the sa	18
5.14	Dereferencing a	19
5.15	Escaping of the a	20
5.16	Conversion of si EOF [signconv]	21
5.17	Use of an implic	22
5.18	Failing to close f [fileclose].....	23
5.19	Failing to detect	24
5.20	Forming invalid pointers by library function [libptr].....	26
5.21	Allocating insufficient memory [insufmem].....	28
5.22	Forming or using out-of-bounds pointers or array subscripts [invptr].....	29
5.23	Freeing memory multiple times [dblfree].....	34
5.24	Including tainted or out-of-domain input in a format string [usrfmt].....	35
5.25	Incorrectly setting and using errno [inverrno].....	37
5.26	Integer division errors [diverr].....	39
5.27	Interleaving stream inputs and outputs without a flush or positioning call [ioileave].....	40
5.28	Modifying string literals [strmod].....	41
5.29	Modifying the string returned by getenv, localeconv, setlocale, and strerror [libmod].....	42
5.30	Overflowing signed integers [intoflow].....	43
5.31	Passing a non-null-terminated character sequence to a library function that expects a string [nonnull]	44
5.32	Passing arguments unsigned char	45
5.33	Passing pointers in parameters [rest]	46
5.34	Reallocating or fre	47
5.35	Referencing uninit	48
5.36	Subtracting or con	49
5.37	Tainted strings are	50
5.38	Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr].....	50
5.39	Using a tainted value as an argument to an unprototyped function pointer [taintnoproto].....	51
5.40	Using a tainted value to write to an object using a formatted input or output function [taintformatio].....	52
5.41	Using a value for fsetpos other than a value returned from fgetpos [xfilepos].....	52
5.42	Using an object overwritten by getenv, localeconv, setlocale, and strerror [libuse].....	53
5.43	Using character values that are indistinguishable from EOF [chreof].....	54
5.44	Using identifiers that are reserved for the implementation [resident].....	55
5.45	Using invalid format strings [invfmtstr].....	57
5.46	Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink [taintsink].....	58

3.2 Coverage Summary

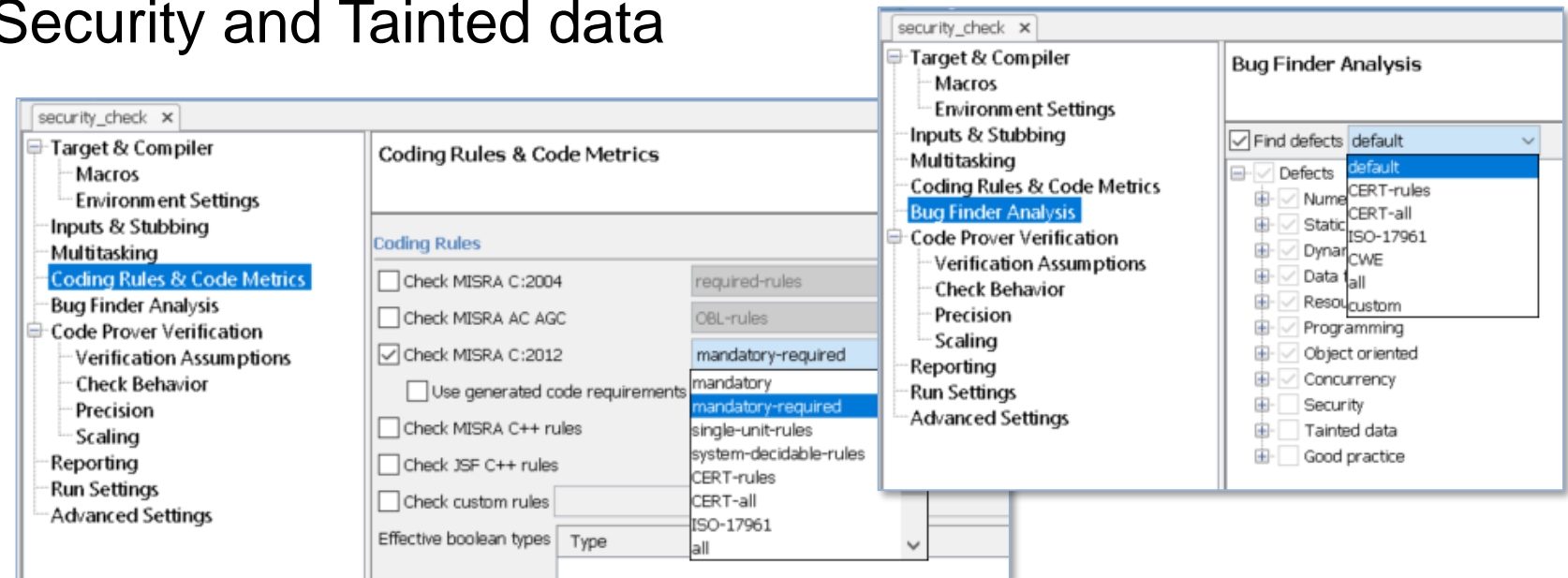
In summary, the coverage of MISRA C:2012 against C Secure is as follows:

Classification	Strength	Number
Explicit	Strong	20
	Weak	2
Implicit	Strong	1
	Weak	6
Restrictive	Strong	11
	Weak	0
Partial/Restrictive	Strong/None	2
None	None	4
Total		46

Polyspace Bug Finder And Security Standard

- Well-known defects for unreliable code like buffer overflows, dead code...
- Plus two categories: Security and Tainted data

- Security Standards
 - CERT-C
 - ISO-17961 (Full)
 - MISRA-C 2012 (Full)
 - CWE



- The mapping table between Polyspace Bug Finder and Security Standard
 - MATLAB_INSTALL\polyspace\resources\Polyspace Results R2018b.xlsx

How does *Polyspace* help you with embedded software security?

- Detecting security vulnerabilities and underlying defects early
- Provides Exhaustive Documentation and recommendation for security fix
- Proving absence of certain critical vulnerabilities
- Complying with industry standards – MISRA-C, CWE, CERT C, ISO 17961

Q & A