PARASOFT®

Zephyr® Project

Developer Summit 2022

June 8-9, 2022

Mountain View, CA + Virtual

Zephyr® Project
Developer Summit 2022

June 8-9, 2022
Mountain View, CA + Virtual

PARASOFT®

lattix

Tools and Methodologies to test compliance with IEC61508

Neil Langmead – *Director of Professional Services – Lattix*

Andrey Madan – *Director of Solution Engineering - Parasoft*

Andrey Madan
Director of Solution Engineering
andrey.madan@parasoft.com

PARASOFT

Neil Langmead
Director of Professional Services
neil.langmead@lattix.com

lattix

Agenda

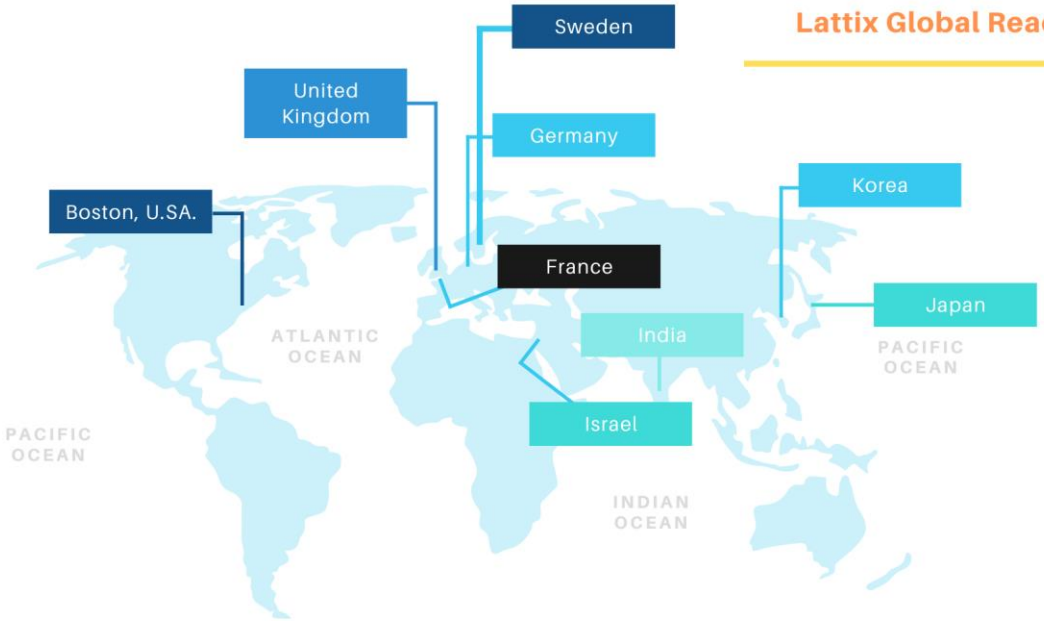
- Exploring the IEC 61508 standard
- Architecture Analysis and methods of monitoring dependencies
- Coding standards for C and C++ and enforcement strategies
- Unit Testing frameworks and approaches
- Code Coverage requirements and methods of meeting them
- Demo: Achieving FuSa compliance in GitLab CI
- Q & A

Parasoft at a Glance

30+ yrs. leading the industry in Continuous Quality and Automated Software Testing

- Focused on software testing tools:
  - C/C++, Embedded Safety Critical: Automotive, MilAero, Medical, Industrial, Rail
  - Java, .NET, Microservices, APIs, End to End: Banking, Financial, Travel.
- Member of the **MISRA** Committee
- **Silver Sponsor Linux Foundation (Zephyr)**

Lattix



- **Services:**
  - Compliance and Functional Safety
  - DevOps automation
  - Refactoring code and build
- **Product: Lattix Architect, Traci**
  - Understand, define (refactor), and control software **architecture**
  - Helps with compliance for IEC 61508, IEC 62304, ISO26262 and DO178C
  - Supported languages: C/C++, C#, Javascript, Java, Python, UML/SysML, Oracle, Fortran, ADA

Functional Safety standard: IEC61508

Architecture design principles in IEC 61508

Table F.2 – Software design and development: software architecture design

(see IEC 61508-3 7.4.3 and IEC 61508-3 Table C.2)

	Property	Definition
2.1	Completeness with respect to Software Safety Requirements Specification	The Software Architecture Design addresses all the safety needs and constraints raised by the Software Safety Requirements Specification.
2.2	Correctness with respect to Software Safety Requirements Specification	The Software Architecture Design provides an appropriate answer to the specified software safety requirements.
2.3	Freedom from intrinsic design faults	The Software Architecture Design and the Design Documentation are free from faults that can be identified independently of any specified software Safety Requirement. Examples: deadlocks, access to unauthorised resources, resource leaks, intrinsic incompleteness (i.e., failure to address all the situations that derive from the design itself).
2.4	Simplicity and understandability. Predictability of behaviour.	The Software Architecture Design allowing a correct and accurate prediction, in all specified situations, of the functioning of the Software. In particular, these situations include erroneous and failure situations. Predictability implies in particular that the functioning does not depend on items that cannot be controlled by designers or users.
2.5	Verifiable and testable design	The Software Architecture Design and the Design Documentation allow and facilitate the production of credible evidence that all the specified software safety requirements are correctly taken into account by the Design and that the Design is free from intrinsic faults. Verifiability may imply derived properties like simplicity, modularity, clarity, testability, provability, etc., depending on the verification techniques used.
2.6	Fault tolerance	The Software Architecture Design gives assurance that the software will have a safe behaviour in the presence of errors (internal errors, errors of operators or of external systems). Defensive design may be active or passive. Active defensive designs may include, features like detection, reporting and containment of errors, graceful degradation and cleaning up of any undesirable side effects prior to the resumption of normal operation. Passive defensive designs include features that guarantee the imperviousness to particular types of errors or particular conditions (avalanches of inputs, particular dates and times) without the software taking any specific action.
2.7	Defence against Common Cause Failure from external events	The Software Architecture Design facilitates the identification of common cause failure modes and effective precautions against failure.

Static Code Analysis to help comply with C.2.6.2

- MISRA C 2012
- AUTOSAR C++ 14
- CERT C /C++
- CWE top 25 2019
- OWASP API Security

C.2.6.2 Coding standards

NOTE This technique/measure is referenced in Table B.1 of IEC 61508-3.

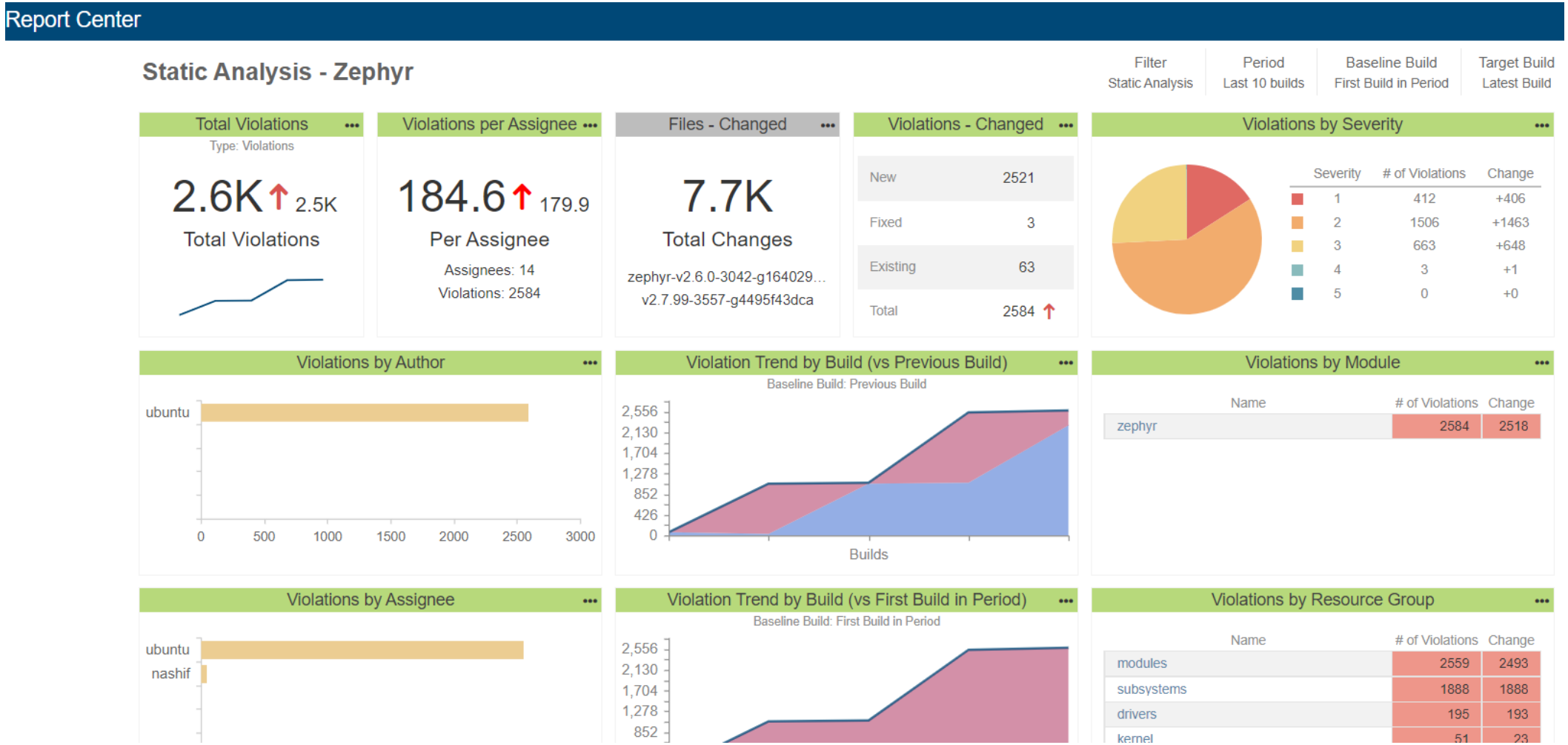
Aim: To reduce the likelihood of errors in the safety-related code and to facilitate its verification.

Description: The following principles indicate how safety-related coding rules (for any programming language) can assist in complying with the IEC 61508-3 normative requirements and in achieving the informative "desirable properties" (see Annex F). Account should be taken of available support tools.

IEC 61508-3 Requirements & Recommendations	*Coding Standards Suggestions*
Modular approach (Table A.2-7, Table A.4-4)	Software module size limit (Table B.9–1) and software complexity control (Table B.9–2). Examples: • Specification of "local" size and complexity metrics and limits (for modules) • Specification of "global" complexity metrics and limits (for overall modules organisation) • Parameter number limit / fixed number of subprogram parameters (Table B.9–4) Information hiding/encapsulation (Table B.9–3): e.g., incentives for using particular language features.

Parasoft C/C++test is being used by Zephyr project

https://docs.zephyrproject.org/latest/contribute/coding_guidelines/index.html



Unit / Module Testing

Table F.5 – Software design and development: software module testing and integration

(see IEC 61508-3 7.4.7 and IEC 61508-3 7.4.8 and IEC 61508-3 Table C.5)

	Property	Definition
5.1	Completeness of testing and integration with respect to the design specifications	The software testing examines the software behaviour sufficiently thoroughly to ensure that all the requirements of the Software Design Specification have been addressed.
5.2	Correctness of testing and integration with respect to the design specifications (succesful completion)	The module testing task is completed, and there exists specific evidence to claim that the safety requirements have been met.
5.3	Repeatability	Consistent results are produced on repeating the individual assessments carried out as part of the module testing and integration.
5.4	Precisely defined testing configuration	The module testing and integration has been applied to the right version of the elements and the software, with the results claimed, and allows the results to be linked to the specific configuration of the “as-built” software.

Unit Testing - FAQ

- What is the value of unit/module testing? Is that just for certification?
- Is the Unit Testing just to get my Code Coverage criteria?
- Can I execute unit tests on the host instead of target? What are the pros and cons?
- What are most popular testing frameworks and tools?
- Do I need a TUV certified tool?

Code Coverage

C.5.8 Structure-based testing

NOTE This technique/measure is referenced in Table B.2 of IEC 61508-3.

Aim: To apply tests which exercise certain subsets of the program structure.

Description: Based on analysis of the program, a set of input data is chosen so that a large (and often prespecified target) percentage of the program code is exercised. Measures of code coverage will vary as follows, depending upon the level of rigour required. In all cases, 100 % of the selected coverage metric should be the aim; if it is not possible to achieve 100 % coverage, the reasons why 100 % cannot be achieved should be documented in the test report (for example, defensive code which can only be entered if a hardware problem arises). The first four techniques in the following list are mentioned specifically in the recommendations in Table B.3 of IEC 61508-3 and are widely supported by testing tools; the remaining techniques could also be considered.

- **Entry point (call graph) coverage**: ensure that every subprogram (subroutine or function) has been called at least once (this is the least rigorous structural coverage measurement).

  NOTE In object-oriented languages, there can be several subprograms of the same name which apply to different variants of a polymorphic type (overriding subprograms) which can be invoked by dynamic dispatching. In these cases every such overriding subprogram should be tested.

- **Statements**: ensure that all statements in the code have been executed at least once.
- **Branches**: both sides of every branch should be checked. This may be impractical for some types of defensive code.
- **Compound conditions**: every condition in a compound conditional branch (i.e. linked by AND/OR) is exercised. See MCDC (modified condition decision coverage, ref. DO178B).
- **LCSAJ**: a linear code sequence and jump is any linear sequence of code statements, including conditional statements, terminated by a jump. Many potential subpaths will be infeasible due to constraints on the input data imposed by the execution of earlier code.
- **Data flow**: the execution paths are selected on the basis of data usage; for example, a path where the same variable is both written and read.
- **Basis path**: one of a minimal set of finite paths from start to finish, such that all arcs are included. (Overlapping combinations of paths in this basis set can form any path through that part of the program.) Tests of all basis path has been shown to be efficient for locating errors.

Requirements Traceability

C.2.11 Traceability

Aim: To maintain consistency between lifecycle stages.

Description: In order to ensure that the software that results from lifecycle activities meets the requirements for correct operation of the safety-related system, it is essential to ensure consistency between the lifecycle stages. A key concept here is that of "traceability" between activities. This is essentially an impact analysis to check (1) that decisions made at an earlier stage are adequately implemented in later stages (forward traceability), and (2) that decisions made at a later stage are actually required and mandated by earlier decisions.

Forward traceability is broadly concerned with checking that a requirement is adequately addressed in later lifecycle stages. Forward traceability is valuable at several points in the safety lifecycle:

- from the system safety requirements to the software safety requirements;
- from the Software Safety Requirements Specification, to the software architecture;
- from the Software Safety Requirements Specification, to the software design;
- from the Software Design Specification, to the module and integration test specifications;
- from the system and software design requirements for hardware/software integration, to the hardware/software integration test specifications;
- from the Software Safety Requirements Specification, to the software safety validation plan;
- from the Software Safety Requirements Specification, to the software modification plan (including reverification and revalidation);
- from the Software Design Specification, to the software verification (including data verification) plan;

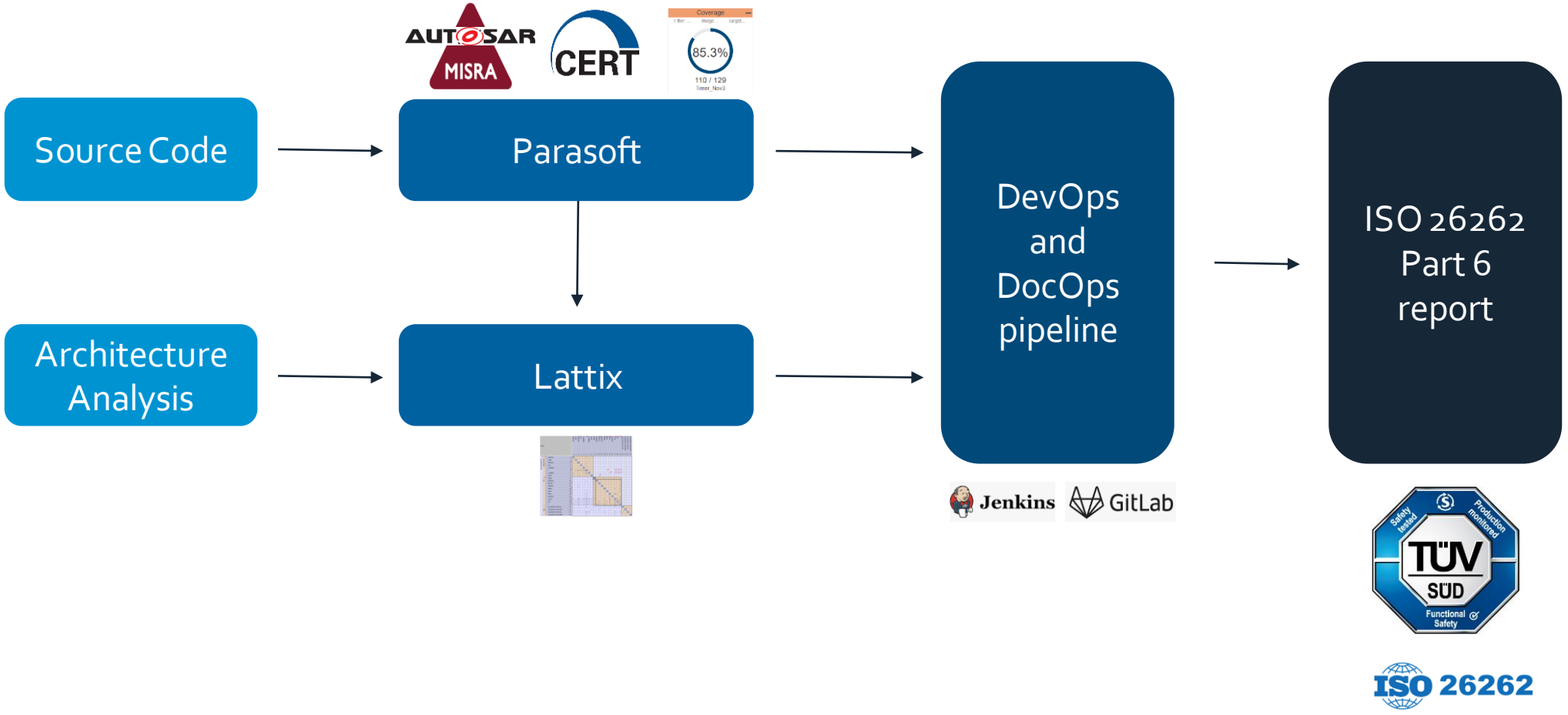
Let’s take a look - Demo

lattix

PARASOFT

The best path to automating compliance testing

Architecture Information + Static analysis + Dynamic test and coverage data



What do you need to do for IEC 61508?

- Architecture analysis and methods of monitoring dependencies and enforcing design principles.
- Coding standards for C and C++ and enforcement strategies.
- Code Coverage requirements and ways to obtain and merge coverage across multiple levels of testing.
- Unit Testing frameworks and approaches
- DocOps: Generate compliance artifacts on every build

SIGN UP for a guided eval:

https://www.lattix.com/free-trial/

https://alm.parasoft.com/parasoft-lattix-guided-demo

CONTACT:

Neil: neil.langmead@lattix.com

Andrey: andrey.madan@parasoft.com

@parasoft, @lattixDSM

@parasoftcorp

@parasoftcorporation

@parasoft, @lattix