Misra Parallelism Safety-critical Guidelines for C++11, 17, Then C++20, 23

ANDREAS WEIS, MICHAEL WONG & ILYA BURYLOV

Cppcon
The C++ Conference
2021
October 24-29

Staff Engineer at Woven Planet

Andreas Weis (he/him)

/ ComicSansMS

@DerGhulbus

Co-organizer of the Munich C++ User Group (MUC++)

Member of WG21 (ISO C++) and MISRA C++

Working on the Runtime framework for the Arene platform at Woven Planet

woven planet

Ilya Burylov

Principle Engineer at Intel

An architect of C++ software solutions for autonomous driving market in Intel

Contribution into functional safety MISRA standard

Contribution into WG21 in threading, vectorization and numerics.

Contribution into SYCL

Distinguished Engineer

- Chair of SYCL Heterogeneous Programming Language
- ISO C++ Directions Group past Chair
- Past CEO OpenMP
- ISOCPP.org Director, VP http://isocpp.org/wiki/faq/wg21#michael-wong
- michael@codeplay.com
- fraggamuffin@gmail.com
- Head of Delegation for C++ Standard for Canada
- Chair of Programming Languages for Standards Council of Canada
  Chair of WG21 SG19 Machine Learning
  Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded
- Editor: C++ SG5 Transactional Memory Technical Specification
- Editor: C++ SG1 Concurrency Technical Specification
- MISRA C++ and AUTOSAR
- Chair of Standards Council Canada TC22/SC32 Electrical and electronic components (SOTIF)
- Chair of UL4600 Object Tracking
- RISC-V Datacenter/Cloud Computing Chair
- http://wongmichael.com/about
- C++11 book in Chinese: https://www.amazon.cn/dp/B00ETOV2OQ

Michael Wong

Argonne and Oak Ridge National Laboratories Award Codeplay® Software to Further Strengthen SYCL™ Support Extending the Open Standard Software for AMD GPUs

17 June 2021

LEMONT, IL, and OAK RIDGE, TN, and EDINBURGH, UK, June 17, 2021 - Argonne National Laboratory (ANL) in collaboration with Oak Ridge National Laboratory (ORNL), has awarded Codeplay a contract implementing the oneAPI DPC++ compiler, an implementation of the SYCL open standard software, to support AMD GPU-based high-performance compute (HPC) supercomputers.

NSITEXE, Kyoto Microcomputer and Codeplay Software are bringing open standards programming to RISC-V Vector processor for HPC and AI systems

29 October 2020

Implementing OpenCL™ and SYCL™ for the popular RISC-V processors will make it easier to port existing HPC and AI software for embedded systems

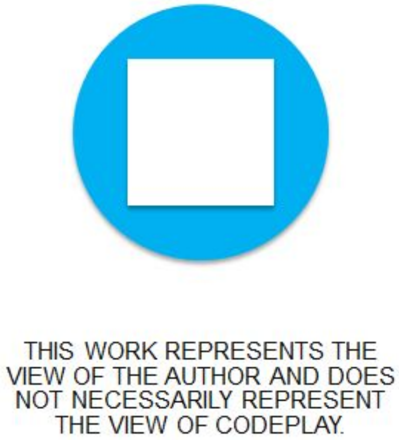
NERSC, ALCF, Codeplay Partner on SYCL for Next-generation Supercomputers

02 February 2021

The **National Energy Research Scientific Computing Center** (NERSC) at **Lawrence Berkeley National Laboratory** (Berkeley Lab), in collaboration with the **Argonne Leadership Computing Facility** (ALCF) at Argonne National Laboratory, has signed a contract with **Codeplay Software** to enhance the LLVM SYCL™ GPU compiler capabilities for NVIDIA® A100 GPUs.

We build GPU compilers for some of the most powerful supercomputers in the world

Acknowledgement and Disclaimer



Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.

Images belong to their respective copyrights.

But I claim all credit for errors, and stupid mistakes. **These are mine, all mine! You can't have them.**

Agenda

1. Current status of C++ safety: MISRA and C++ CG
2. Parallel Safety rules
3. Automotive Safety case

Safety Critical API Evolution

New Generation Safety Critical APIs for Graphics, Compute and Display

minimize API surface area , reduce ambiguity, UB, increase determinism

Industry Need
for CPU/GPU Acceleration APIs designed to ease system safety certification

Rendering Compute Display

RISC-V

SCSC
FOR EVERYONE WORKING IN SYSTEM SAFETY
WWW.SCSC.UK

ISO International Organization for Standardization

UNECE WP.29

C++

SYCL

ISO 26262

ISO/PAS 21448

UL 4600

MISRA

SAE INTERNATIONAL

ISO/IEC JTC 1/SC 42
Artificial intelligence

IBM

- ISO/IEC AWI TR 20547-1 [Under development]
  Information technology -- Big data reference architecture -- Part 1: Framework and application process
- ISO/IEC TR 20547-2:2018
  Information technology -- Big data reference architecture -- Part 2: Use cases and derived requirements
- ISO/IEC DIS 20547-3 [Under development]
  Information technology -- Big data reference architecture -- Part 3: Reference architecture
- ISO/IEC TR 20547-5:2018
  Information technology -- Big data reference architecture -- Part 5: Standards roadmap
- ISO/IEC AWI 22989 [Under development]
  46T46T46TArtificial Intelligence Concepts and Terminology
- ISO/IEC AWI 23053 [Under development]
  Framework for Artificial Intelligence (AI) Systems Using Machine Learning (ML)

Source: https://www.iso.org/committee/6794475/x/catalogue/p/1/u/1/w/0/d/0

Participation Details for Italy Here: http://www.uninfo.it/index.php

Many Safety Critical APIs

- Misra: checkable rules only
- Autosar C++ Guidelines: a mix of meta guidelines and checkable rules
- High Integrity C++: for static checkers
- WG23 Programming Vulnerabilities: for team leads
- C++ Core Guidelines: a mix
- C++ Study Group 12 Vulnerabilities: for standards
- C Safe and Secure Study Group: for standards
- Carnegie Mellon Cert C and C++: a mix
- Joint Strike Fighter ++: checkable rules
- Common Weakness Enumeration: a mix

- Khronos Safety Critical Advisory Forum
- OpenCL/SYCL Safety Critical
- Vulkan Safety Critical
- JTC1/SC42 Machine Learning WG3 Trustworthiness
- ITC22/SC32 SOTIF WG8 SOTIF, WG13, WG14
- SAE ORAD
- UL4600
- RISC-V Safety/Security

Which one to choose and what is the difference?

- Safe but not C++11/14/17/20
  - Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, 2005
    - With the help of Bjarne Stroustrup
  - **MISRA C++:2008 Guidelines for the use of the C++ language in critical systems, The Motor Industry Software Reliability Association, 2008**
    - **Continues to be the reference despite its age**
    - **For automated static analysis tools**
    - **Aimed for embedded domains**
- C++11/14/17/20 but not safe
  - High Integrity C++ Coding Standard Version 4.0, Programming Research Ltd, 2013
    - Some parallelism
  - Software Engineering Institute CERT C/C++ Coding Standard, Software Engineering Institute Division at Carnegie Mellon University, 2016
    - Most recent effort based on C 11 and C++ 14
  - C++ Core Guidelines, http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines, 2017
    - Most recent effort based on C++17 + 20
    - An excellent style guide for greater elegance, and safety/performance
    - No specific domains, also for static analysis and guidance
  - WG23 Vulnerabilities ISO for C, C++, Ada, Fortran, …
    - Guidelines for teamleads
    - Reviewed with each ISO C, C++, Ada, Fortran help

Comparing coding standards

Coding Standard	**C++ Versions**
Autosar	C++14
Misra	C++03 (working to C++17)
High Integrity CPP	C++11
JSF	C++03
C++ CG	C++11/14/17/20/latest
CERT C++	C++14

Pedigree

Coding Standard	Number of Rules	Number of rules in common with Autosar			% of rules in common
		Identical	Small Diff	Big Diff	
Misra C++	229	138	38	32	91%
High Integrity C++	155	0	99	25	80%
JSF ++	226	0	143	28	76%
C++ CG	412	0	174	49	54%
CERT C++	156	0	75	33	69%

These are the 2 most important guidelines today

C++ CG: Meta + automated checkable rule

- **Follow Bjarne's talk on type+resource Safety on C++ CG**
- **Aim for bug free code with high performance and elegant coding style**
- **Meta rules + checkable rules**
  - USe GSL, CLion,
- Use a carefully crafted set of programming techniques
  - supported by library facilities
  - enforced by static analysis.
- Available on GitHub
  - https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

- Philosophy
  - **Express ideas directly in code**
  - **Write in ISO Standard C++**
  - **Express intent**
  - **Ideally, a program should be statically type safe**
  - **Prefer compile-time checking to run-time checking**
  - **What cannot be checked at compile time should be checkable at run time**
  - **Catch run-time errors early**
  - **Don't leak any resources**
  - **Don't waste time or space**
  - **Prefer immutable data to mutable data**
  - **Encapsulate messy constructs, rather than spreading through the code**

MISRA and Automated checkable rules with some meta

"First, do no harm. After that, go nuts."

MISRA – Motor Industry Software Reliability Association

MISRA C

- 1998 - Guidelines for the use of the C language in vehicle based software
  - MISRA C:1998 (MISRA C1)
- 2004 - MISRA C:2004 Guidelines for the use of the C language in critical systems
  - MISRA C:2004 (MISRA C2)
- 2013 - MISRA C:2012 Guidelines for the use of the C language in critical systems
  - MISRA C:2012 (MISRA C3)
  - 159 rules of which 138 are statically enforceable

MISRA C++

- 2008 - Guidelines for the use of the C++ language in critical systems
  - 228 rules of which 219 are statically enforceable

- Code will always have bug, but they must do no harm

What is still missing?

So far most only deal with Sequential code

Very few deal with Parallel code

Even fewer deal with Concurrent, event driven code

None deal with Heterogeneous dispatch code

There is always going to be:

- Dirty data, faulty HW, integrity problems
- NEED Freedom from interference, which is much harder in multithread system
- Heterogeneous-> AI/ML safety

https://docs.google.com/document/d/14E0BYqsH_d7fMKvXvaZWoNWtIC65cYBw0aZp4dlev0Q/edit

MISRAC++ParallelConcurrencyHeteroRulesOverview

File Edit View Insert Format Tools Add-ons Help Last edit was made seconds ago by Michael Wong

Outline

Note: this is an early draft WIP. It's known to be incomplet and incorrekt, and it has lots of badformatting.

Table of Content

Table of contents

Heading numbers format 1.2.3
Display until level 6

Always open for this document

Powered by Josué MOËNS @ LumApps

Stage 1: extensive deep analysis of 81 rules

- Started in 2019 at a MISRA meeting
  - Why are there no rules for parallelism in MISRA?
- 2019-2021: Phase 1 complete
  - Reviewed 81 rules pulled from
    - C++CG
    - HIC++
    - REphrase H2020 project
    - CERT C++
    - JSF++ (no parallel rules)
    - WG23 (no parallel rules)
    - Added some from our own contributions
- Many joined, average 5-8 per meeting
  - Also consulted outside concurrency and safety experts
- Shared Drive of Phase 1 analysis:
  - https://docs.google.com/document/d/14E0BYqsH_d7fMKvXvaZWoNWtIC65cYBw0aZp4dlev0Q/edit#heading=h.yt0hxah53p9e

MISRAC++ParallelConcurrencyHeteroRulesStatusPhase1

A6 | 0.3.6 [5] Do not use std::thread

	Rule	Category	decidable via human review	decidable via tools	status	Destination: Tools vs C++ Core guideline	Reason for keeping and
2	***0.2.2 [2] Do not use platform specific multi-threading facilities***	advisory	easy	partially	consider later		only partially detectable, e.g
3	***0.3.1 0.3.x [82] Make std::threads unjoinable on all paths***	advisory	complex	yes, on system level	consider later		use [7] instead
4	***0.3.4 [3] A thread shall not access objects whose lifetime has expired***	required	complex	partially	accept for initial revision	CP23 is high level vs more specific	it may exclude certain techr
5	***0.3.5 [4] Thread callable object may receive only global or static objects via pointer or***	?mandatory	easy	partially	consider later		complex behavior of detach
6	***0.3.6 [5] Do not use std::thread***	advisory	easy	yes, on local level	accept for initial revision	CP 25 is different	straightforward and decidab
7	***0.3.8 [7] Do not call std::thread::detach() function? Join on all Available exit paths***	required	easy	yes, on local level	accept for initial revision	CP 26	better than [82] in decidabili
8	***0.3.9 [8] Verify resource management assumptions of std::thread with the implementat***	directive	complex	no	consider later		directive - keep directive for
9	***0.4.1 [9] Do not call member functions of std::mutex, std::timed_mutex, std::recursive_***	required	easy	yes, on local level	accept for initial revision	CP 20 is close, we are a little more specific but it is t	straightforward and decidab
10	***0.4.3 [11] Use std::lock(), std::try_lock() or std::scoped_lock to acquire multiple mutex***	required	easy	yes, on local level	accept for initial revision	based on CP21	straightforward and decidab
11	***0.4.4 [12] Do not destroy objects of the following types std::mutex, std::timed_mutex, s***	mandatory	complex	yes, on system level	accept for initial revision	NO CG, but not good for CG as it is a clear error, no	clear UB related
12	***0.4.5 [13] Mutexes locked with std::lock or std::try_lock shall be wrapped with std::loc***	required	easy	yes, on local level	accept for initial revision	NO CG, might be good for CG	straightforward and decidab
13	***0.4.6 [14] Do not call virtual functions and callable objects passed by argument of the***	advisory	complex	yes, on system level	consider later		
14	***0.4.8 [16] Objects of std::lock_guards, std::unique_locks, std::shared_lock and std::sc***	required	easy	yes, on local level	accept for initial revision	based on CP24 add shared_lock	straightforward and decidab
15	***0.4.9 [17] Define a mutex together with the data it guards. Use synchronized_value<T>***	directive	complex	no	consider later		related API is not yet confir
16	***0.4.11 [19] There shall be no code path which results in locking of the non-recursive m***	mandatory	complex	yes, on system level	accept for initial revision	no CG, but hard for human, so nard for CG	clear UB related
17	***0.4.12 [20] The order of nested locks unlock shall form a DAG***	required	complex	yes, on system level	accept for initial revision	no CG, but hard for human, so hard for CG	should enspire tools detecti
18	***0.4.13 [21] std::recursive_mutex and std::recursive_timed_mutex should not be used***	advisory	easy	yes, on local level	accept for initial revision	good for CG, good for MISRA	a sign of too complex soluti
19	***0.4.14 [22] There should be a code path, where at least one member functions is callec***	advisory	easy	yes, on system level	drop		not a safety concern
20	***0.5.1 [23] std::condition_variable::wait, std::condition_variable::wait_for, std::conditio***	required	easy	yes, on local level	accept for initial revision	already in CG	straightforward and decidab
21	***0.5.3 [25] std::conditional_variable::notify_one() can be used if all threads must perform the same set of oper***		?	?	consider later		
22	***0.5.4 [26] Do not use std::condition_variable_any on a std::mutex***	advisory	easy	yes, on local level	accept for initial revision	good for CG/tools	straightforward and decidab
23	***0.6.1 [27]Use only std::memory_order_seq_cst for atomic operations***	required	easy	yes, on local level	accept for initial revision	good for CG/tools, more specific then CGF dont use	straightforward and decidab
24	***0.7.1 [28] Use a future to return a value from a concurrent task***		?	?	drop		hardly formalizable
25	***0.7.2 [29] Use an async() to spawn a concurrent task***		?	?	drop		to be replace with [5]
26	***0.8.1 [30] Don't try to use volatile for synchronization***		?	?	drop		to be replace with [32]
27	***0.8.2 [31] Use volatile only to talk to non-C++ memory***		?	?	drop		should not be in scope of pa
28	**0.8.3 [32] Volatile variables shall not be accessed from different threads.**	required	complex	may be, on system level	accept for initial revision	good for tools, meta for CG CP8	should enspire tools detecti
29	***0.9.1 [33] Bit-fields of the same object, which are not separated by not-bit-field or zero***	required	complex	may be, on system level	consider later		very small use case
30	***0.9.2 [34] Synchronize access to data shared between threads using a single lock***	advisory	complex	may be, on system level	consider later		not perfectly formalizable

Лист1

Rule decidability

- Human review
  - Generally simple rules
  - Code snippets
  - Basic syntax matches intention
- Automated tool
  - Static scope: can be convoluted but doable and simple for this generation of tools
  - Dynamic scope: much more complex, hard even for tools of this generation, may be doable with whole program analysis
    - Intention is hidden
- Both Human and Automated tools
  - Generally simple cases
  - Intention is shown in syntax
- Neither are good
  - Very hard cases, dynamic scope, whole program analysis
  - Intention is not clear
    - In these cases we wonder if an [[intention:]] attribute might help

Where should parallel/concurrency/hetero rules go?

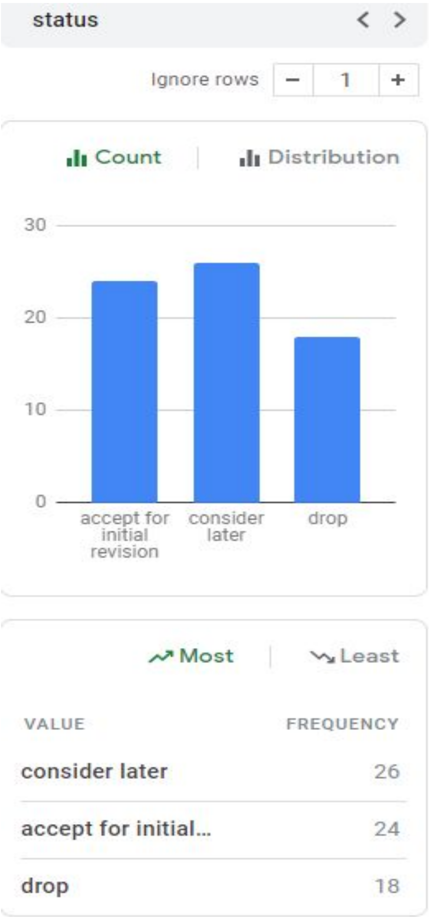
Human decidable	**Tool decidable**	**Suitable tools in order of preference**
Easy	Easy	C++CG, MISRA tools
Easy	Hard	C++CG, Tools will be meta or undecidable, lots of false positive May be bad rule for tools
Hard	Easy	MISRA tools, CG Meta
Hard	Hard	Neither, META directive; Code guidelines Obvious rules, but hard to verify Might not be a good rule anyway Need a new [[intention::]] attribute

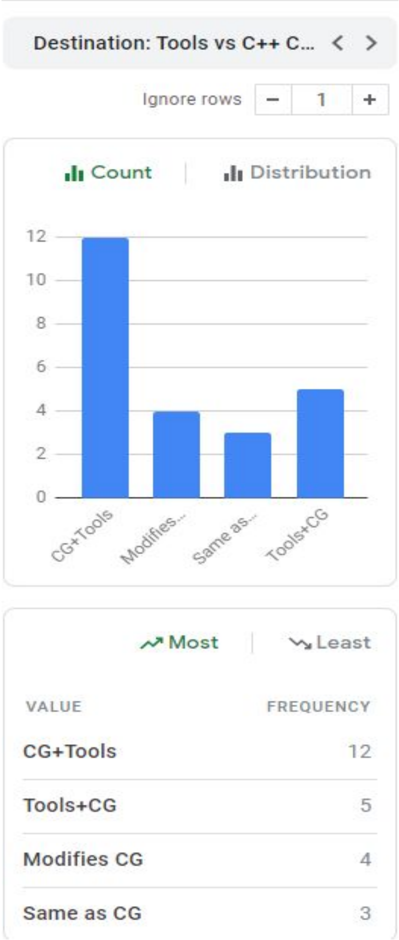
Stage 2: collate

- Category
  - Mandatory: 8
  - Required: 12
  - Advisory: 12
  - Directive: 5
- Decidable by humans
  - Easy: 27
  - Medium: 1
  - Complex: 20
  - Unknown yet: 9
- Decidable via automated tools
  - Yes, on a local level: 20
  - Yes, on a system level: 6
  - Maybe, on a system level: 7
  - No: 8
  - Unknown yet: 11

CG, Misra, both or neither

- Accepted: for initial entry 24
    - CG+tools: 12
    - Tools+CG: 5
    - Modifies CG: 4
    - Same as CG: 3
  - Deferred for future: 26
  - Rejected: 18
- Shared drive of Status from Phase 1:
  - https://docs.google.com/spreadsheets/d/1f-NX2z6axIyv5P0mh4aeNfKO7KLSVSTtZrTwS2YO02M/edit#gid=0





Agenda

1. Current status of C++ safety: MISRA and C++ CG
2. Parallel Safety rules
3. Automotive Safety case

For humans and tools -> C++CG and MISRA

- Rule 13 Mutexes locked with std::lock or std::try_lock shall be wrapped with std::lock_guard, std::unique_lock or std::shared_lock with adopt_lock tag within the same scope

The rule intention is to employing RAII for controlling the state of mutexes in exceptional conditions

Is it easy to detect via review?

- just check std::lock arguments

Good for C++CG

Is it easy to detect by tool?

- tool can check std::lock arguments

Good for MISRA

For humans and tools -> C++CG and MISRA

- Rule 13 Mutexes locked with std::lock or std::try_lock shall be wrapped with std::lock_guard, std::unique_lock or std::shared_lock with adopt_lock tag within the same scope

C++CG has these rules:

CP.20: Use RAII, never plain `lock()`/`unlock()`

Reason Avoids nasty errors from unreleased locks.

CP.21: Use `std::lock()` or `std::scoped_lock` to acquire multiple `mutex`es

Reason To avoid deadlocks on multiple `mutex`es.

Rule 13 is intentionally friendlier for tools, if compared with CP.21

For humans not tools -> C++CG

- Rule 39 Use std::call_once to ensure a function is called exactly once (rather than the Double-Checked Locking pattern)

The rule intention is to avoid common errors, which might be introduced if common concurrency task is being reproduced with less care

Is it easy to detect via review?

- via careful understanding of the coder intention

Good for C++CG

Is it easy to detect by tool?

- It is difficult to detect incorrectly written Double-Checked Locking pattern

Not so good for tools

For humans not tools -> C++CG

- Rule 39 Use std::call_once to ensure a function is called exactly once (rather than the Double-Checked Locking pattern)

C++CG has these rules:

CP.110: Do not write your own double-checked locking for initialization

Reason Since C++11, static local variables are now initialized in a thread-safe way. When combined with the RAII pattern, static local variables can replace the need for writing your own double-checked locking for initialization. std::call_once can also achieve the same purpose. Use either static local variables of C++11 or std::call_once instead of writing your own double-checked locking for initialization.

For tools not humans -> MISRA

- Rule 19 The order of nested locks/unlock shall form a DAG

The rule intention is to avoid deadlocks via careful tracing of locking and unlocking order.

Is it easy to detect via review?

- Starting from the moderately complex code, it becomes very difficult to trace the order of locks

Harder for C++CG

Is it easy to detect by tool?

- It is theoretically possible, if all the code underneath the specified block is visible

Better for Tools

For humans only -> meta rules

- Rule 8 Verify resource management assumptions of std::thread with the implementation of standard library of choice

Safety implies careful analysis of assumption introduced by dependencies, this one should be reviewed with especial care

Is it easy to detect via review?

- It is not visible in code and should be applied on higher review levels

Too high level for C++CG?

Is it easy to detect by tool?

- It is not visible in code and is not visible for code analysis tools

Directive for tools/MISRA

Agenda

1. Current status of C++ safety: MISRA and C++ CG
2. Parallel Safety rules
3. Automotive Safety case

Why Concurrency guidelines for automotive?

- ISO26262 lists concurrency aspects as one topic to be covered by “Modeling and coding guidelines”
- But should a safety-critical system contain concurrent control-flow at all?

- Typical embedded systems are small and follow static execution patterns
- Even a multi-core automotive chip may have parallel execution but no concurrency if components scheduled on different cores do not interact

The Old World vs. The New

- Traditionally, automotive systems rely on static scheduling
- Each task is given a predetermined time slice in the schedule for execution
- The complete schedule is configured upfront as part of the system design
- Temporal execution of tasks is completely deterministic

⇒ Tasks not scheduled to run in parallel will not overlap. Synchronization between cores is handled by a thin basic software module layer.

⇒ Component interaction across cores is minimized

The Old World vs. The New

- Static scheduling works well only if the number of components is small or the interaction between components is minimal
- With compute-intensive applications like highly automated driving, parallel computation is needed to process data in time

Old: Many small independent applications sharing compute resources of a single CPU. Simple basic software layer at the bottom.

New: One single application requiring all compute resources of a powerful multi-core CPU and possibly a number of auxiliary hardware accelerators. Fully fledged OS at the bottom.

⇒ Lots of concurrency, asynchronous APIs as the default

Asynchronous APIs in Adaptive Autosar

```
ara::com::FindServiceHandle find_service_handle = RadarServiceProxy::StartFindService(
  [](ara::com::ServiceHandleContainer<RadarServiceProxy::HandleType> handles, ara::com::FindServiceHandle) {









  });
// ...
RadarServiceProxy::StopFindService(find_service_handle);
```

Asynchronous APIs in Adaptive Autosar

```
ara::com::FindServiceHandle find_service_handle = RadarServiceProxy::StartFindService(
  [](ara::com::ServiceHandleContainer<RadarServiceProxy::HandleType> handles, ara::com::FindServiceHandle) {
    if (handles.empty()) { return; }
    RadarServiceProxy service(handles.front());

    ara::core::Future<uint32_t> fut = service.UpdateRate.Get();






    // ...
  });
// ...
RadarServiceProxy::StopFindService(find_service_handle);
```

Asynchronous APIs in Adaptive Autosar

```
ara::com::FindServiceHandle find_service_handle = RadarServiceProxy::StartFindService(
  [](ara::com::ServiceHandleContainer<RadarServiceProxy::HandleType> handles, ara::com::FindServiceHandle) {
    if (handles.empty()) { return; }
    RadarServiceProxy service(handles.front());

    ara::core::Future<uint32_t> fut = service.UpdateRate.Get();
    auto fut2 = fut.then([](ara::core::Future<uint32_t> f) -> auto {
      uint32_t update_rate = f.get();
      // ...
    });

    // ...
  });
// ...
RadarServiceProxy::StopFindService(find_service_handle);
```

Bug Example

```
struct SharedData {

  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {

  shared_data.update_rate = f.get();


});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  shared_data.mtx.lock();
  shared_data.update_rate = f.get();


});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  shared_data.mtx.lock();
  shared_data.update_rate = f.get();
// WARNING: Do not destroy objects of type std::mutex
//          if object is in locked state
});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  shared_data.mtx.lock();
  shared_data.update_rate = f.get();
  shared_data.mtx.unlock();

});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  shared_data.mtx.lock();
  shared_data.update_rate = f.get();  // get() may throw!
  shared_data.mtx.unlock();

});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  shared_data.mtx.lock();              // WARNING: Do not call member functions of std::mutex
  shared_data.update_rate = f.get();  // get() may throw!
  shared_data.mtx.unlock();

});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  std::scoped_lock{ shared_data.mtx };
  shared_data.update_rate = f.get();


});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  std::scoped_lock{ shared_data.mtx };  // WARNING: Objects of type std::scoped_lock shall always be named
  shared_data.update_rate = f.get();


});
```

Bug Example

```
struct SharedData {
  std::mutex mtx;
  uint32_t update_rate;
} shared_data;


fut.then([&shared_data](ara::core::Future<uint32_t> f) -> auto {
  std::scoped_lock lk{ shared_data.mtx };
  shared_data.update_rate = f.get();


});
```

Does this mean concurrency is now fine for safety-critical software?

Rules only attempt to catch common pitfalls in using concurrency facilities.

They are only one building block in a larger safety strategy.

Safety implications of use of concurrency must be carefully evaluated in the context of the overall safety strategy

Final Words

More safety:Parallel/concurrency for C++11, 14, 17, C++20

	Asynchronous Agents	**Parallel collections**	**Mutable shared state**	**Heterogeneous/Distributed**
abstractions from C++11, 14, 17, 20	C++11: thread,lambda function, TLS, async C++ 20: Jthreads +interrupt _token, coroutines	C++11: packaged tasks, promises, futures, C++ 17: ParallelSTL, control false sharing C++20 : Vec execution policy, Algorithm un-sequenced policy	C++11: locks, memory model, mutex, condition variable, atomics, static init/term, C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence, C++ 17: scoped _lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies C++20: atomic_ref, Latches and barriers, atomic<shared_ptr> Atomics & padding bits Simplified atomic init Atomic C/C++ compatibility Semaphores and waiting Fixed gaps in memory model , Improved atomic flags, Repair memory model	C++11: lambda C++14: generic lambda C++17: , progress guarantees, TOE, execution policies C++20: atomic_ref

Future safety rules for C++ 20/23 parallelism

- Not inventing, just documenting common wisdom which takes time
- MISRA NEXT is really MISRA 2008 + AUTOSAR
    - Need more manpower
    - Need more experience on the safety of new features
    - Will not cover C++20 and might even miss a few C++17 features
- MISRA parallel will also be in stages
    - C++11 atomics, async .mm
    - C++14 shared lock
    - C++17 parallel algo, futures, (still need more deep dive) unseq,
    - C++20 latches barriers, coroutine, atomic ref,
    - C++23 senders and receivers?
    - C++26 executors networking?, concurrency TS2?

Conclusion and Future plan

- 2021: plan to integrate with MISRA 202X NEXT release
  - 17 rules to MISRA C++ NEXT
  - 17 rules to C++CG
  - 4 CG rules to be modified
  - Reset Deferred to next phase
- 2022-?
  - Work on Deferred rules
  - Add new rules covering
    - Coroutines, parallel algorithm, executors,
  - Aim for next release of MISRA NEXT+ CG NEXT
- Continue with more phases and more releases

C++ Will need to integrate safety with ML

From sequential->concurrency

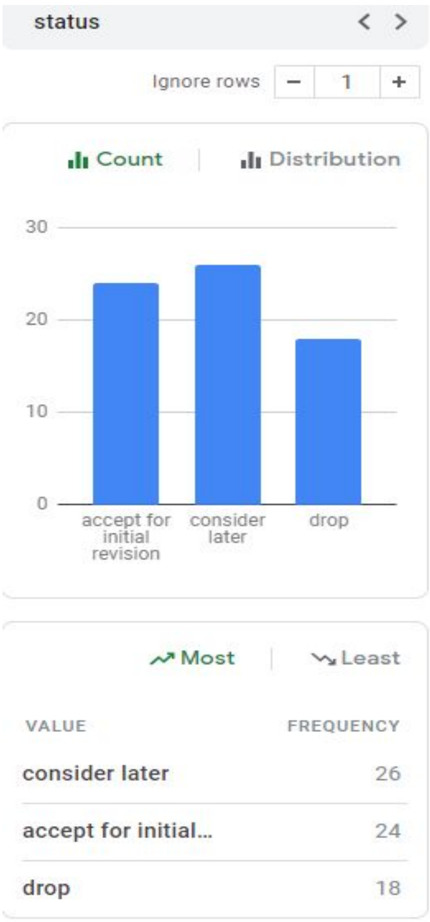
- 26262
- Adaptive autosar

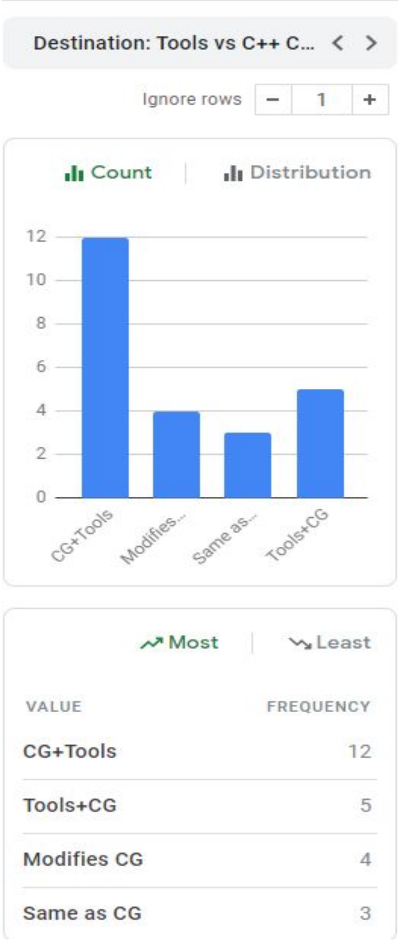
From concurrency->heterogeneous

- ML /AI trustworthiness, safety
- 21448 SOTIF
- UL4600
- SAE ORAD

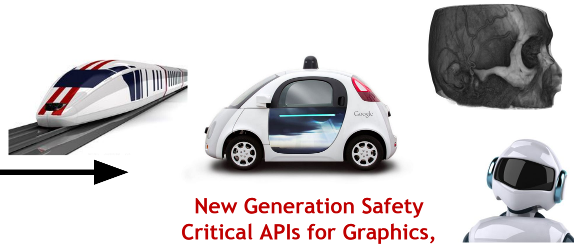
CG, Misra, both or neither

- Accepted: for initial entry 24
    - CG+tools: 12
    - Tools+CG: 5
    - Modifies CG: 4
    - Same as CG: 3
  - Deferred for future: 26
  - Rejected: 18
- Shared drive of Status from Phase 1:
  - https://docs.google.com/spreadsheets/d/1f-NX2z6axIyv5P0mh4aeNfKO7KLSVSTtZrTwS2YO02M/edit#gid=0





Safety Critical API Evolution



New Generation Safety Critical APIs for Graphics, Compute and Display

minimize API surface area , reduce ambiguity, UB, increase determinism

Industry Need
for CPU/GPU Acceleration APIs designed to ease system safety certification

Rendering Compute Display

RISC-V

SCSC FOR EVERYONE WORKING IN SYSTEM SAFETY WWW.SCSC.UK

ISO International Organization for Standardization

UNECE WP.29

SYCL

ISO 26262

ISO/PAS 21448

UL 4600

MISRA

SAE INTERNATIONAL

ISO/IEC JTC 1/SC 42
Artificial intelligence

- ISO/IEC AWI TR 20547-1 [Under development]
  Information technology -- Big data reference architecture -- Part 1: Framework and application process
- ISO/IEC TR 20547-2:2018
  Information technology -- Big data reference architecture -- Part 2: Use cases and derived requirements
- ISO/IEC DIS 20547-3 [Under development]
  Information technology -- Big data reference architecture -- Part 3: Reference architecture
- ISO/IEC TR 20547-5:2018
  Information technology -- Big data reference architecture -- Part 5: Standards roadmap
- ISO/IEC AWI 22989 [Under development]
  46T46T46TArtificial Intelligence Concepts and Terminology
- ISO/IEC AWI 23053 [Under development]
  Framework for Artificial Intelligence (AI) Systems Using Machine Learning (ML)

Source: https://www.iso.org/committee/6794475/x/catalogue/p/1/u/1/w/0/d/0

Participation Details for Italy Here: http://www.uninfo.it/index.php