Apex.AI®

Safe and certified software for autonomous mobility

How Apex.AI Certified ROS 2 According to ISO 26262 ASIL-D

Dejan Pangercic (dejan@apex.ai)

Relevance to Autoware Community

1. Autoware.Auto uses ROS2. If AWF members want to certify whole or parts of Autoware.Auto and then sell that as part of the fully certified complete SW AD stack - they need a certified framework

2. ROS2 is a C++14 product. This talk is largely about how to certify any C++ code base (independent of ROS2, Autoware.Auto, ...)

3. The process of certification improved the quality of Apex.AI fork of ROS2. Consequently this talk is also about the code quality

4. ISO 26262 is at the center of standardization in the automotive industry. There is no more important standard and one can not avoid it if you build production systems for automotive. Consequently this talk is about sharing an experience of going through the process of ISO 26262 certification

But the entire AD Stack is Huge

Focus of this talk

Full Stack

Product	User Application	System Integration	Application	Release		
	Algorithmic Software	2D+3D Perception, Tracking, Class.	AI/ML based Scene Understanding	Localization	Route Planning, Decision Making	Motion Planning and Control
On-Board Software	Framework Software	Abstraction of hardware, middleware, OS, interfaces	Middleware	Safety Support	Security Support	Diagnostics
	Operating System	Kernel	Scheduler	Driver		
	Functional Safety	Regulations, Homologation	Security	Safety	Verification	Validation
Methodologies	Testing	Unit tests	Regression Tests	Integration Tests	SIL/HIL Tests	Vehicle Tests
	Design	User Experience	Interior Interaction Design	Exterior Interaction Design		
	Development Tools	Development Environment	Visualization	Simulation		
	Data Processing	Data Recording	Data Playback	Data Annotation	Data Management and Analytics	
Off-Board Software and Data	Operations	Fleet Management	Fleet Routing	Tele-Operation	Diagnostics	OTA Update
	Mapping	Creation	Annotation	Update	Distribution	
	Maps	Topological Layer	Relational Layer	Physical Layer		
	Sensors	GNSS	INS/IMU	Lidar	Camera	Radar
Hardware Stack	Components	Computing	Cloud Connectivity	In-Vehicle Connectivity	Data Recorder	Data Storage
	Vehicle	Interfaces to:	Drive Train	Braking System	Steering System	Electronics

AVP

We have a Base Functional AD System - What Next?

1. **System Safety:**
   a. ISO 26262 Certification
      i. Code
      ii. HW
   b. System specification and operating environment (ODD)
   c. HARA
   d. Design for redundancy
   e. Validation plan
      i. System validation (ISO 15288)
   f. SOTIF
      i. Scenario-based testing with statistical sampling in simulation (NCAP, NHTSA scenarios)
   g. Closed course testing
   h. Public road testing
   i. Simulation
      i. SIL and HIL Testing

2. **AV Technology:**
   a. Object and event detection and response
   b. Fallback systems

3. **AV Operation:**
   a. ODD
   b. AV Operators
   c. Incident response and management

4. **Interfaces:**
   a. Passenger and road user interface
   b. Cybersecurity
   c. Data management

Source: Motional, [VSSA, 2021]

We have a Base Functional System - What Next?

Focus of the rest of the talk

1. **System Safety:**
   a. ISO 26262 Certification
      i. Code
      ii. HW
   b. System specification and operating environment (ODD)
   c. HARA
   d. Design for redundancy
   e. Validation plan
      i. System validation (ISO 15288)
   f. SOTIF
      i. Scenario-based testing with statistical sampling in simulation (NCAP, NHTSA scenarios)
   g. Closed course testing
   h. Public road testing
   i. Simulation
      i. SIL and HIL Testing

2. **AV Technology:**
   a. Object and event detection and response
   b. Fallback systems

3. **AV Operation:**
   a. ODD
   b. AV Operators
   c. Incident response and management

4. **Interfaces:**
   a. Passenger and road user interface
   b. Cybersecurity
   c. Data management

Source: Motional, VSSA, 2021

Apex.OS Development Lifecycle

ISO 26262/SEooC/part3,part6…. processes

Automotive Stakeholder Require-ments (ASR)

Requirements	Architecture	Unit Design	V&V	Conf. Reviews
Elicitation, Safety Concept, SW Safety Requirements	UML (unified modeling language), FMEA	SCA (Static Code Analysis), SW practices outline, coverage, FMEA	Req, arch, unit, integration, system, performance, fault injection tests	Safety manual, Restrictions, Traceability

builtin_interfaces_cert
connext_micro_support_cert
allocator_cert
logging_cert
rclcpp_cert
threading_cert
Apex_ecu_monitor (native)
Apex_utils (native)

ROS

builtin_interfaces
connext_micro_support
allocator
logging
rclcpp
threading

Feature set reduction

Apply real-time and determinism constraints

1. Memory static
2. Remove blocking calls and recursions

Apex.OS Cert

Apex.OS Certification Activities per package

1. Reduce the feature set of a package and extensions
2. Investigation to make APIs memory static & ensure no blocking calls
3. Static Code Analysis (SCA)
4. Structural Coverage (statement, branch and MC/DC)
5. Notations of Designs (modelling diagrams)
6. Principles of SW architecture and design
7. Control and data flow analysis
8. Integration and Specialized tests
9. Requirements and traceability
10. Safety Analysis (FMEA)
11. Generate Safety Artifacts (TUV submission)
12. Testing on Target platform/hardware
13. Tool Classification and Qualification

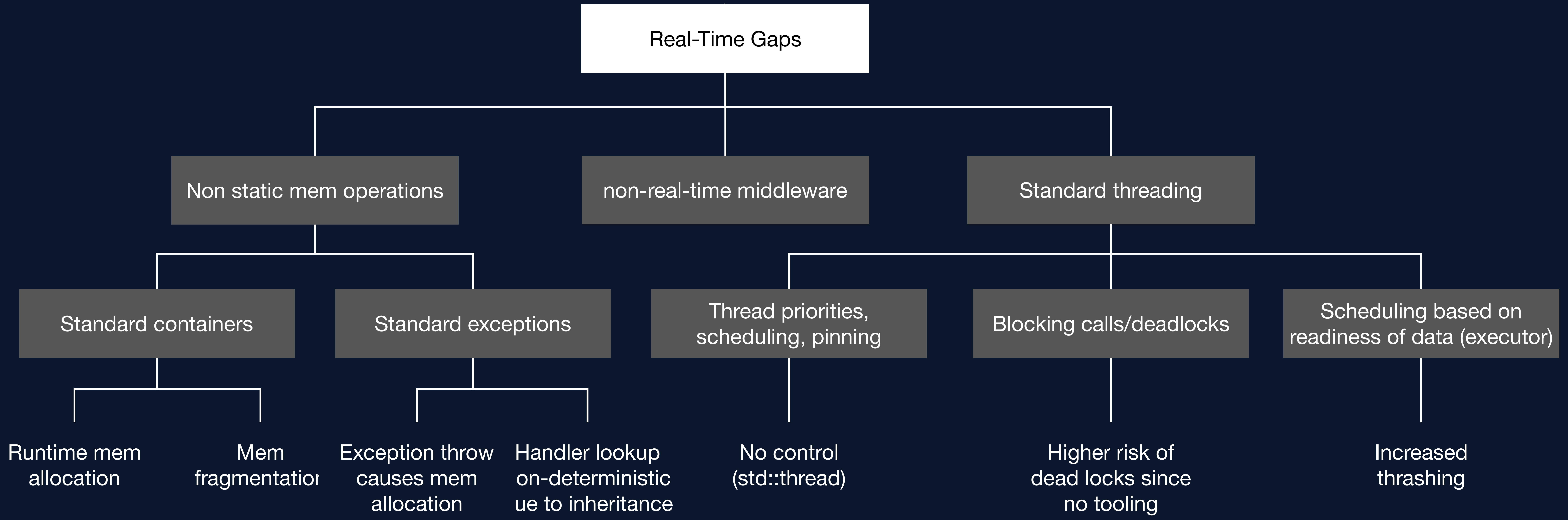
Total 24 pkgs selected for first release of **Apex.OS**

Some activities such as Tool classification and qualification, integration testing done at **Apex.OS** level.

Close to 100 safety artifacts had to be generated to provide evidence of ASIL D compliance to our certification agency.

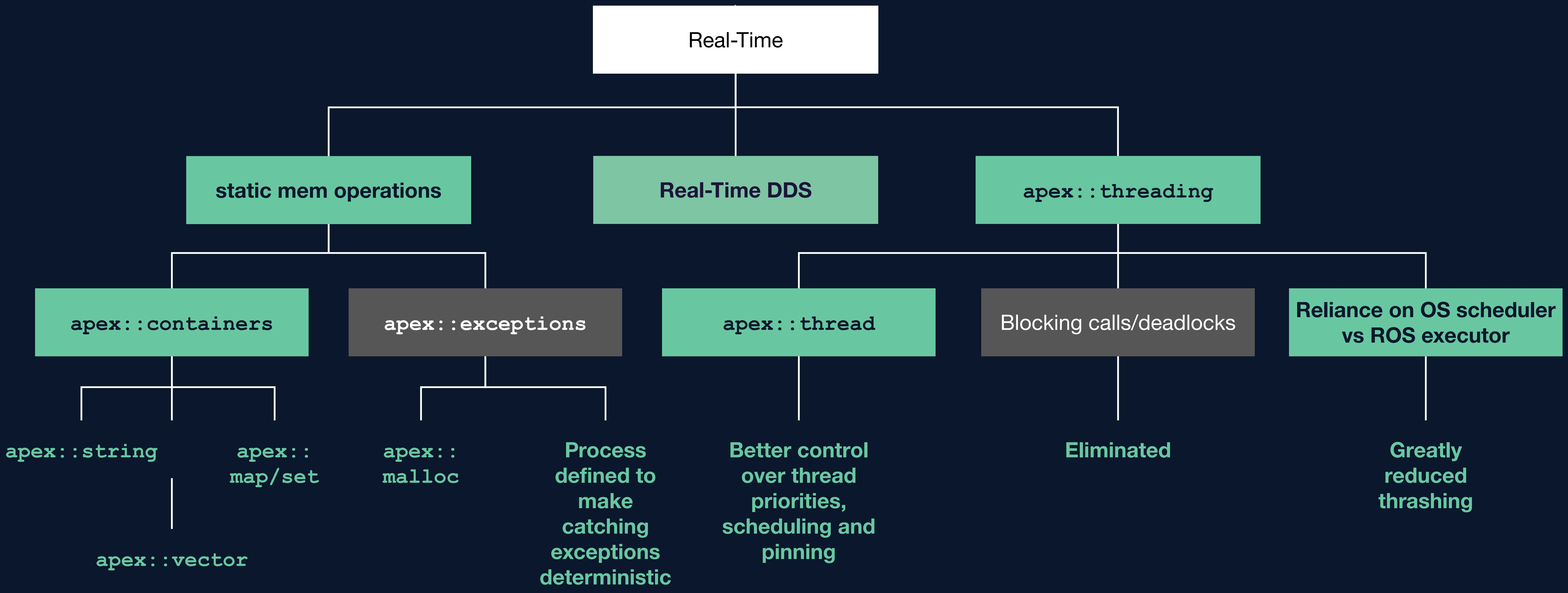
Real-Time Gaps in ROS 2

ROS 2 exhibits the following gaps to enable **real-time performance.**



Apex.OS Solution

Apex.OS addresses the following gaps **to achieve real-time performance.**



Apex.OS Certification Activities per package

1. Reduce the feature set of a package and extensions
2. Investigation to make APIs memory static & ensure no blocking calls
3. Static Code Analysis (SCA)
4. Structural Coverage (statement, branch and MC/DC)
5. Notations of Designs (modelling diagrams)
6. Principles of SW architecture and design
7. Control and data flow analysis
8. Integration and Specialized tests
9. Requirements and traceability
10. Safety Analysis (FMEA)
11. Generate Safety Artifacts (TUV submission)
12. Testing on Target platform/hardware
13. Tool Classification and Qualification

Total 24 pkgs selected for first release of **Apex.OS**

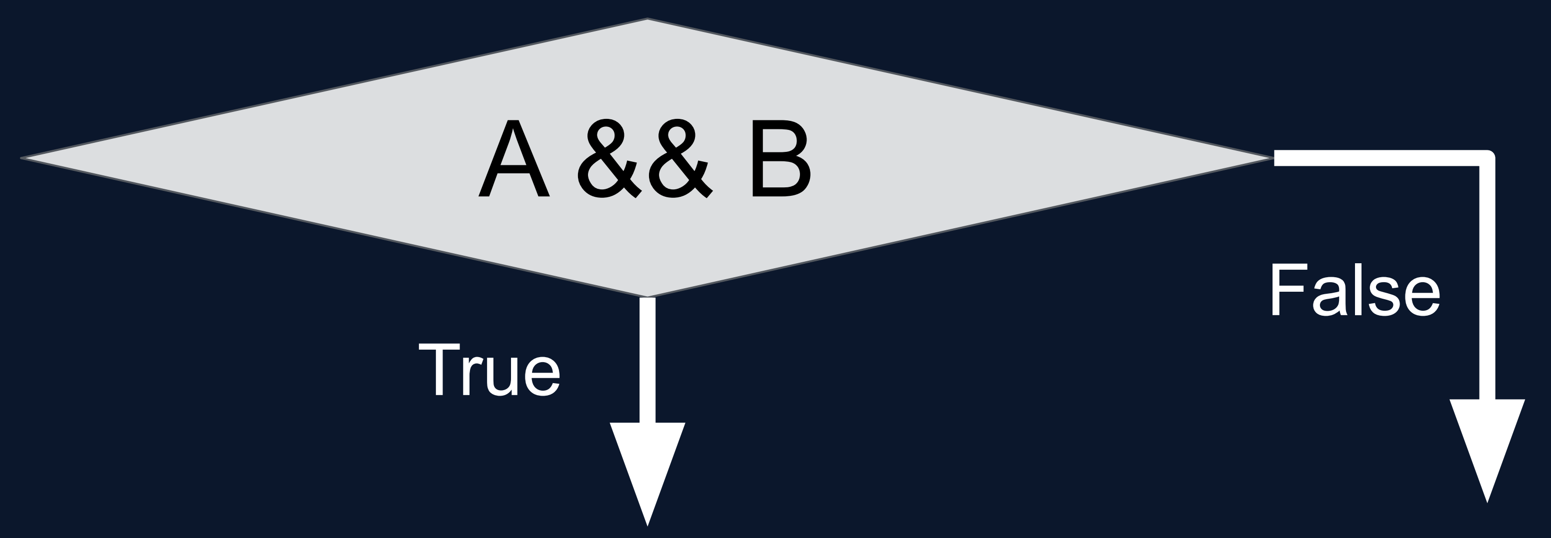
Some activities such as Tool classification and qualification, integration testing done at **Apex.OS** level.

Close to 100 safety artifacts had to be generated to provide evidence of ASIL D compliance to our certification agency.

Branch coverage vs. MC/DC coverage

Branch coverage

Each branch (True and False) should be tested at least once



MC/DC coverage

Every condition in a decision (True and False) should be tested independently

For example (A && B),

1. Create the truth table

ROW	A	B	Res
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

2. Find pairs for which only one condition independently affect the outcome

   a = {1,3}, b= {1,2} -> 1, 2, 3 condition should be tested.

3. For n conditions we only require n+1 tests

Structural Coverage

	Statements	Branches	Pairs
ALL	76516/76516 (100%)	74723/74723 (100%)	13356/13356 (100%)
GNU_Native_7.x_7.5_C++14	76516/76516 (100%)	74723/74723 (100%)	13356/13356 (100%)
os-cert-v1.2.0rc5	76516/76516 (100%)	74723/74723 (100%)	13356/13356 (100%)
allocator	5145/5145 (100%)	5021/5021 (100%)	911/911 (100%)
allocator_gtest	5145/5145 (100%)	5021/5021 (100%)	911/911 (100%)
apex_ecu_monitor	815/815 (100%)	678/678 (100%)	155/155 (100%)
apex_ecu_monitor_gtest	815/815 (100%)	678/678 (100%)	155/155 (100%)
apex_init	66/66 (100%)	65/65 (100%)	9/9 (100%)
apex_init_gtest	66/66 (100%)	65/65 (100%)	9/9 (100%)
apex_malloc	277/277 (100%)	243/243 (100%)	45/45 (100%)
apex_malloc_gtest	277/277 (100%)	243/243 (100%)	45/45 (100%)
apexcpp	2687/2687 (100%)	3208/3208 (100%)	625/625 (100%)
apexcpp_gtest	2687/2687 (100%)	3208/3208 (100%)	625/625 (100%)
apexutils	897/897 (100%)	1271/1271 (100%)	311/311 (100%)
apexutils_gtest	897/897 (100%)	1271/1271 (100%)	311/311 (100%)
baseline_cert_msgs	5244/5244 (100%)	6476/6476 (100%)	1036/1036 (100%)
baseline_cert_msgs_gtest	5244/5244 (100%)	6476/6476 (100%)	1036/1036 (100%)
connext_micro_support dds_dire	34337/34337 (100%)	28245/28245 (100%)	4928/4928 (100%)
apex_dds_cert_gtest	34337/34337 (100%)	28245/28245 (100%)	4928/4928 (100%)
containers	477/477 (100%)	321/321 (100%)	30/30 (100%)
containers_gtest	477/477 (100%)	321/321 (100%)	30/30 (100%)
cpputils	3702/3702 (100%)	5146/5146 (100%)	911/911 (100%)
cpputils_gtest	3702/3702 (100%)	5146/5146 (100%)	911/911 (100%)
launcher	1344/1344 (100%)	1425/1425 (100%)	274/274 (100%)
launcher_gtest	1344/1344 (100%)	1425/1425 (100%)	274/274 (100%)
logging	99/99 (100%)	44/44 (100%)	4/4 (100%)
logging_gtest	99/99 (100%)	44/44 (100%)	4/4 (100%)
rcl	1477/1477 (100%)	1807/1807 (100%)	378/378 (100%)
rcl_gtest	1477/1477 (100%)	1807/1807 (100%)	378/378 (100%)
rclcpp	10067/10067 (100%)	9280/9280 (100%)	1593/1593 (100%)
rclcpp_gtest	10067/10067 (100%)	9280/9280 (100%)	1593/1593 (100%)
rcutils	1939/1939 (100%)	2192/2192 (100%)	477/477 (100%)
rcutils_gtest	1939/1939 (100%)	2192/2192 (100%)	477/477 (100%)
rmw	231/231 (100%)	268/268 (100%)	52/52 (100%)
rmw_gtest	231/231 (100%)	268/268 (100%)	52/52 (100%)
settings	5601/5601 (100%)	6972/6972 (100%)	1256/1256 (100%)
settings_gtest	5601/5601 (100%)	6972/6972 (100%)	1256/1256 (100%)
system_monitor	748/748 (100%)	751/751 (100%)	139/139 (100%)
system_monitor_gtest	748/748 (100%)	751/751 (100%)	139/139 (100%)
threading	1363/1363 (100%)	1310/1310 (100%)	222/222 (100%)
threading_gtest	1363/1363 (100%)	1310/1310 (100%)	222/222 (100%)

To get to the 100% of line (statement), branch and MC/DC (pairs) test coverage we had to add 3000 tests (on top of the 1500 existing tests).

What was tedious?

- Getting 100% MC/DC coverage for heavily templated modern C++ code is tedious.
  - Commercial coverage tool has issues, while it parses modern C++ codes such as a lambda function and template code. (e.g. on the next slide)
- The code base has a lot of hard to reach defensive type coding.
  - Required significant stubbing/mocking of C++ standard library, middleware, and external functions that are implemented in Apex.OS. (e.g. on the next slide)

Issues Parsing Certain Modern C++ Constructs

- **A method with multiple lambda functions**
  - The commercial coverage tool could not parse a method that contained multiple lambda functions.

    Solution: fixing the bug of the commercial coverage tool.

```
class Sub{};
class Pub{};
class C
{
public:
  Sub* create_sub();
  Pub* create_pub();
};


void testme(C* ptr)
{
  C* node = new C;
  auto get_sub = [&node] { return node->create_sub(); };
  auto get_pub = [&node] { return node->create_pub(); };
}
```

- **Template code with locally defined class**
  - The commercial coverage tool cannot parse a class internally defined in a function or the class that is used for the parameter of the template class or function.

    Solution: defining the class with the global scope

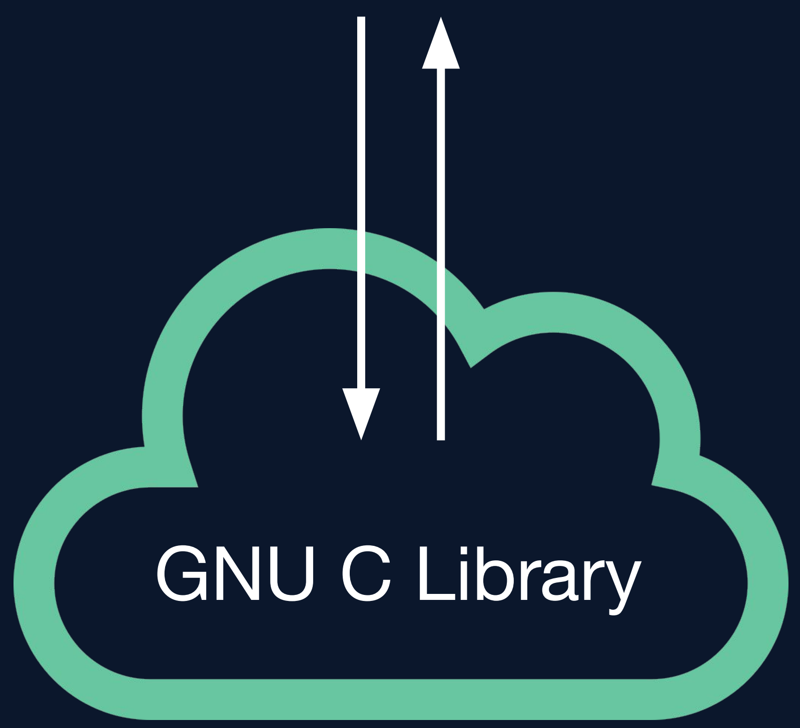
```
int main()
{
    class A {};
    f<A>();
}
```

Mocking GNU C Lib Functions: Example: clock_gettime( )

```
rcutils_system_time_now(rcutils_time_point_value_t * now)
{
  RCUTILS_CHECK_ARGUMENT_FOR_NULL(now,
RCUTILS_RET_INVALID_ARGUMENT);
  struct timespec timespec_now;
  int32_t posix_error;
  posix_error = clock_gettime(CLOCK_REALTIME, &timespec_now);
  if (posix_error != 0) {
    RCUTILS_SET_ERROR_MSG("clock_gettime error");
    return RCUTILS_RET_ERROR;
  }
  if (RCUTILS_WOULD_BE_NEGATIVE(timespec_now.tv_sec,
timespec_now.tv_nsec)) {
    RCUTILS_SET_ERROR_MSG("unexpected negative time");
    return RCUTILS_RET_ERROR;
  }
```

Apex.OS source code

clock_gettime(CLOCK_REALTIME, &timespec_now)



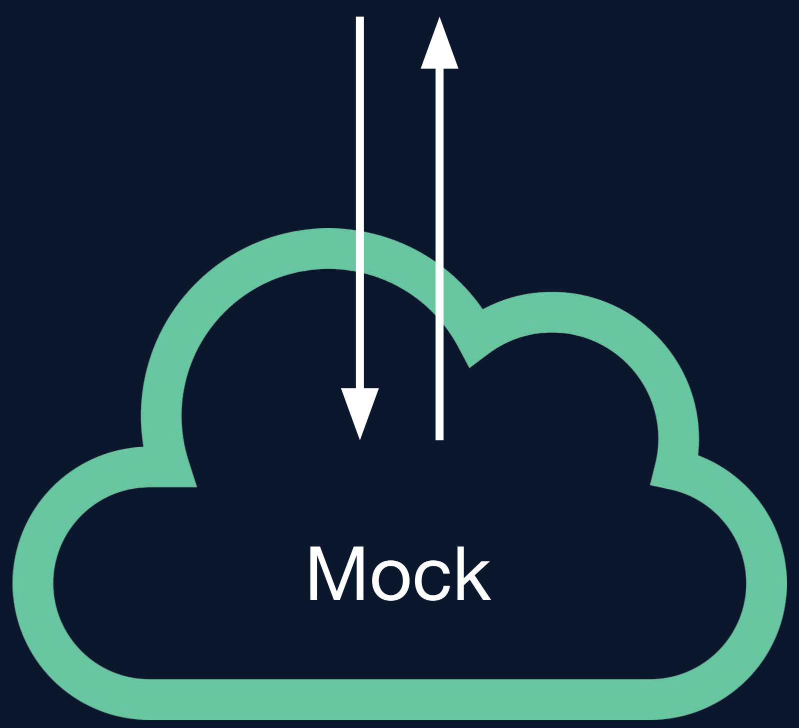

```
int clock_gettime(clockid_t clk_id, struct timespec * tp) __THROW__
{
  int ret = 0;
  if (nullptr != timeUnixPtr) {
    ret = timeUnixPtr->clock_gettime(clk_id, tp);
  }
  return ret;
}

TEST_F(time_gmock, rcutils_system_time_now) {
  rcutils_time_point_value_t now = 0;
  rcutils_ret_t ret;
  EXPECT_CALL(*timeUnixPtr, clock_gettime(_,
_)).WillRepeatedly(Return(-1));
  ret = rcutils_system_time_now(&now);
  EXPECT_EQ(ret, RCUTILS_RET_ERROR);
  rcutils_reset_error();

}
```

Apex.OS test code

clock_gettime(CLOCK_REALTIME, &timespec_now)



Apex.OS Certification Activities per package

1. Reduce the feature set of a package and extensions
2. Investigation to make APIs memory static & ensure no blocking calls
3. Static Code Analysis (SCA)
4. Structural Coverage (statement, branch and MC/DC)
5. Notations of Designs (modelling diagrams)
6. Principles of SW architecture and design
7. Control and data flow analysis
8. Integration and Specialized tests
9. Requirements and traceability
10. Safety Analysis (FMEA)
11. Generate Safety Artifacts (TUV submission)
12. Testing on Target platform/hardware
13. Tool Classification and Qualification

Total 24 pkgs selected for first release of **Apex.OS**

Some activities such as Tool classification and qualification, integration testing done at **Apex.OS** level.

Close to 100 safety artifacts had to be generated to provide evidence of ASIL D compliance to our certification agency.

What was technically challenging?

- There are no good commercial tools for identifying runtime memory allocations and blocking calls.
  We created new internal tool (apex_tracing_check) that uses LTTng framework to flag infractions.

- Making exceptions handling memory static is complex (and still a research topic)
  We solved it by patching system malloc() and a special (exception handling) memory pool.
  (see next slide)

Elimination of Memory Allocation and Blocking Calls (MA/BC) - Approach

- We implemented *apex_tracing_check* tool which is based on LTTng
- The code is instrumented by adding the macro on top of the function
- It requires to be build with some extra compilation flags to enable the macro
- After this, test cases are executed to find infractions

```
bool
Context::sleep_for(const std::chrono::nanoseconds & nanoseconds)
{
  TRACING_CHECK_START()
  std::unique_lock<apex::time_limit_mutex<>> lock(interrupt_mutex_);
  // See this comment for more detail
  // https://gitlab.apex.ai/ApexAI/grand_central/-/merge_requests/3871#note_483889
  (void)interrupt_condition_variable_.wait_for(lock, nanoseconds,
    [this]() {
      m_is_sleeping = true;
      // if context not valid we should return immediately
      return !this->is_valid();
    });
  m_is_sleeping = false;
  // return true if the condition variable was interrupted and did not timeout
  return !this->is_valid();
}
```

- Example on how *apex_tracing_check* will detect and report infraction/s

```
[ RUN      ] TestInit.sleep_for
error: found determinism infractions
infraction: lttng_ust_libc:pthread_mutex_lock_req @ 1604694850601034509
        rclcpp::Context::sleep_for(std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&) :: /home/neil.langmead/8197/apex_ws/build/rclcpp/librclcpp.so
        TestInit_sleep_for_Test::TestBody() :: ./build/rclcpp/test_init
        void testing::internal::HandleExceptionsInMethodIfSupported<testing::Test, void>(testing::Test*, void (testing::Test::*)(), char const*) :: ./build/rclcpp/test_init
        ?
        ?
        ?
        testing::internal::UnitTestImpl::RunAllTests() :: ./build/rclcpp/test_init
        bool testing::internal::HandleExceptionsInMethodIfSupported<testing::internal::UnitTestImpl, bool>(testing::internal::UnitTestImpl*, bool (testing::internal::UnitTestImpl::*)(), char const*) :: ./
build/rclcpp/test_init
        testing::UnitTest::Run() :: ./build/rclcpp/test_init
        main :: ./build/rclcpp/test_init
        __libc_start_main :: /lib/x86_64-linux-gnu/libc.so.6
unknown file: Failure
C++ exception with description "found determinism infractions" thrown in the test body.
[  FAILED  ] TestInit.sleep_for (86083 ms)
```

Example of mutex infraction in sleep_for() function

Elimination of Memory Allocation and Blocking Calls (MA/BC) - Findings

- Using *apex_tracing_check* and having 100% MC/DC coverage, it's possible to verify that there is no MA/BC in runtime.

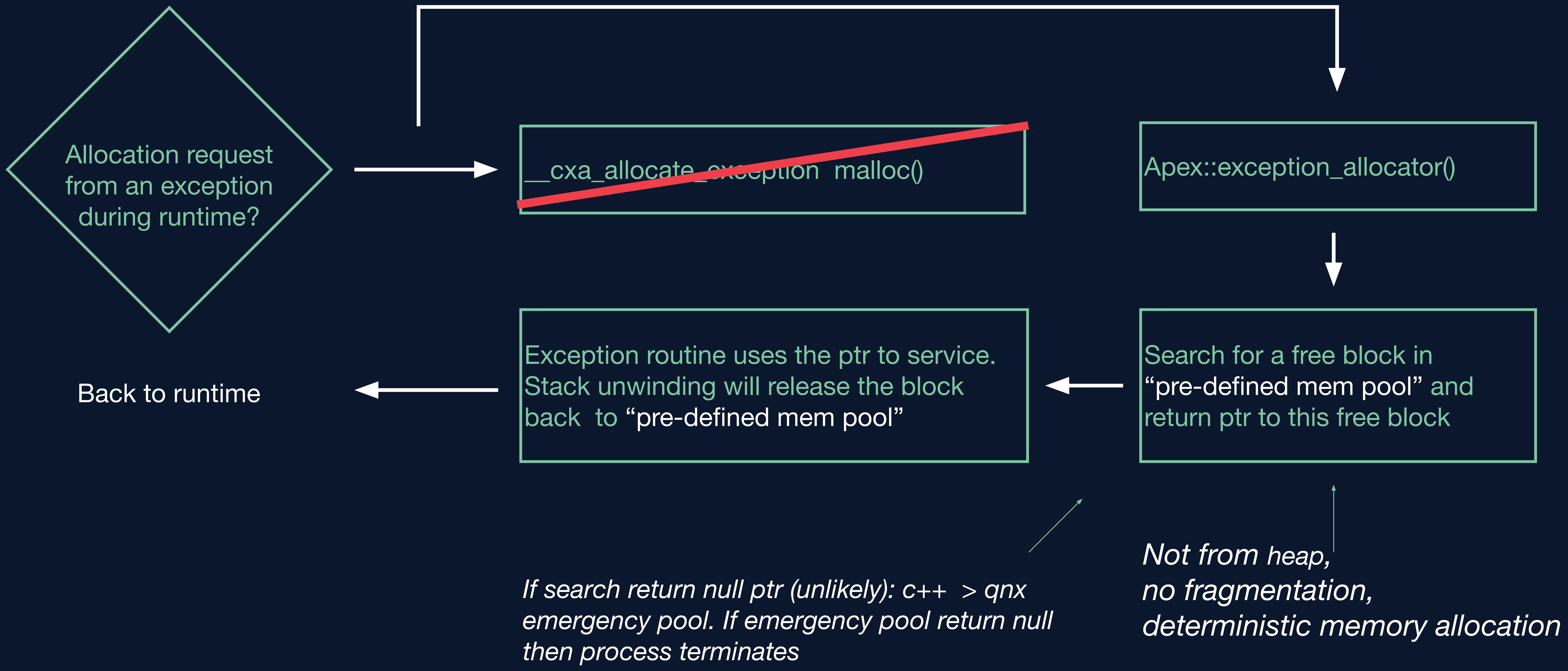
```
  if (ret != RMW_RET_OK) {
-     const auto msg = std::string("failed to compare gids: ") +
-       std::string(rmw_get_error_string().str);
+     const apex::string256_t msg = apex::varargs_to_string("failed to compare gids: ",
+         rmw_get_error_string().str);
    rmw_reset_error();
-     throw std::runtime_error(msg);
+     throw apex::runtime_error(&msg);
  }
```

```
Context::sleep_for(const std::chrono::nanoseconds & nanoseconds)
{
  TRACING_CHECK_START()
-  std::unique_lock<std::mutex> lock(interrupt_mutex_);
+  std::unique_lock<apex::time_limit_mutex<>> lock(interrupt_mutex_);
```

Replacing std::string and std::exception to avoid memory allocations in runtime

Replacing std::mutex with apex::time_limit_mutex to avoid blocking system call in runtime

Allocation Handling during Exception (apex_malloc pkg)



Apex.OS Certification Activities per package

1. Reduce the feature set of a package and extensions
2. Investigation to make APIs memory static & ensure no blocking calls
3. Static Code Analysis (SCA)
4. Structural Coverage (statement, branch and MC/DC)
5. Notations of Designs (modelling diagrams)
6. Principles of SW architecture and design
7. Control and data flow analysis
8. Integration and Specialized tests
9. Requirements and traceability
10. Safety Analysis (FMEA)
11. Generate Safety Artifacts (TUV submission)
12. Testing on Target platform/hardware
13. Tool Classification and Qualification

Total 24 pkgs selected for first release of **Apex.OS**

Some activities such as Tool classification and qualification, integration testing done at **Apex.OS** level.

Close to 100 safety artifacts had to be generated to provide evidence of ASIL D compliance to our certification agency.

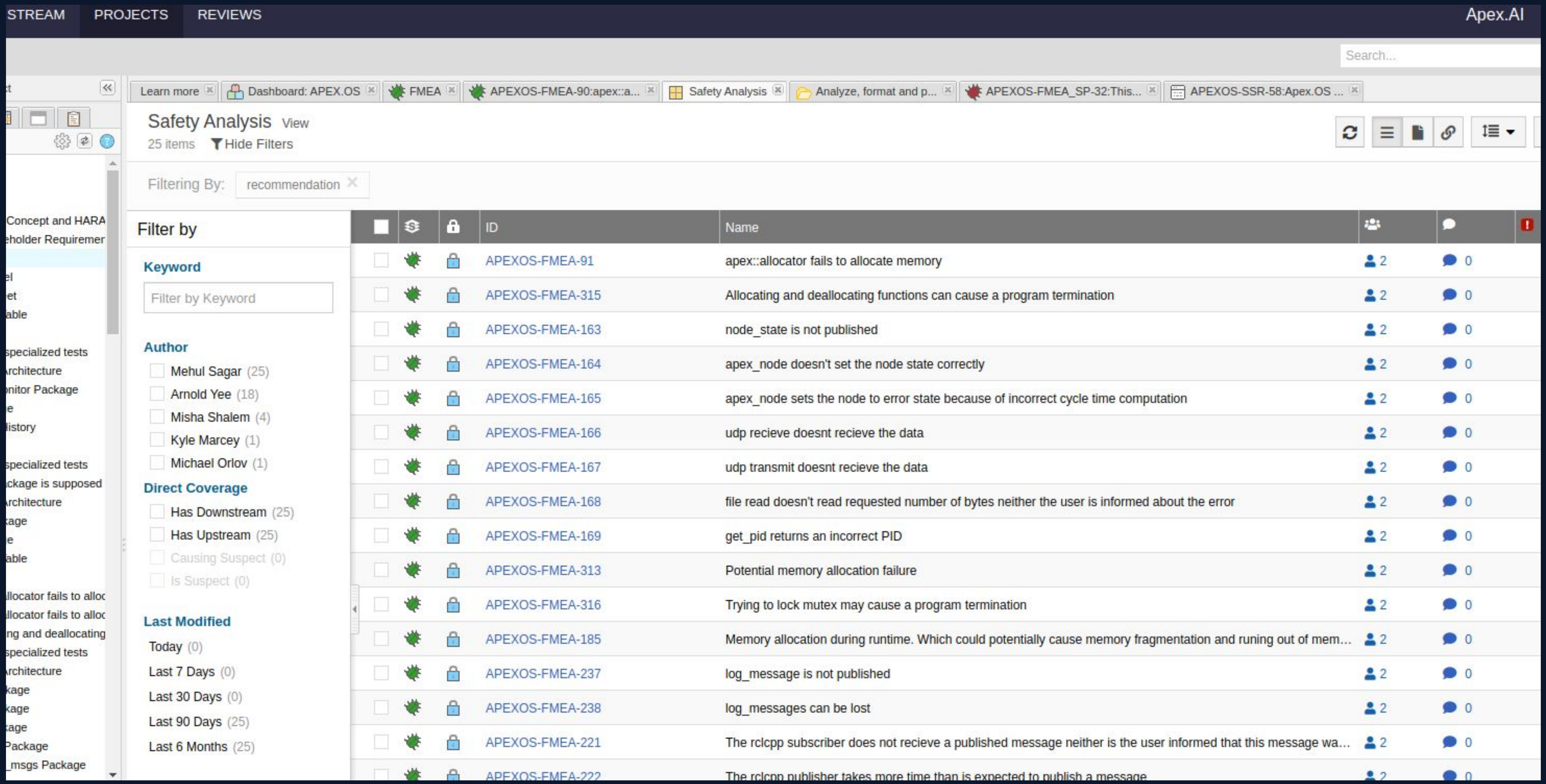
FMEA

FMEA (Failure Mode and Effects) Analysis was performed on each public API in every Cert package.

New tests were added as a result of FMEA=> add example bug from cpputils

33 Software Safety Requirements were added as a result of FMEA Activity.

40 Restrictions and 25 Recommendations added to Safety Manual because of FMEA.



Activity	# Real Issues Found	# Files Changed	# Merge Requests	Git Commits	# Changed code lines
FMEA Analysis	17	55	17	3	62

Examples of Safety Related Changes from ROS 2 to Apex.OS

Example of requirement tracing

“The function rclcpp::Context::sleep_for() shall timeout immediately when zero and negative values are given for the nanoseconds argument”

Solution: The function was not working as described. During the requirement tracing the function and its corresponding tests were fixed

“rclcpp shall provide functionality to assert the liveliness of a publisher”

Solution: There were no tests verifying these requirements. As a result of the analysis a test was added

Example of issue detected as a result from FMEA

“If rclcpp publisher takes more time than is expected to publish a message the application could malfunction”

Solution: If used along with the ApexNode, function calls that exceed the expected time may cause max_cycle_time to be exceeded, which will then notify the user of the failure.

Certification in Numbers

- First round of **Apex.OS** Cert contained **~65K lines of code**
- **14 person years of effort** (1 full time for 2 years, 12 full time for a year)
- 24 ROS 2 + native **Apex.OS** packages certified
- **> $5M cost** in tool licenses, infrastructure, and engineering resources
- **100% statement**, branch, and MC/DC coverage
- **~3000 new tests added** to fulfill safety/certification compliance
- **~300 safety requirements generated** from FMEA, TSC, and Tools C&Q
- **~100 artifacts submitted** to third party auditor (TÜV NORD) for ISO 26262 ASIL D compliance assessment (~2000 A4 pages if printed)
    - **Total of 5 iterations** of audits were conducted by TÜV NORD

Summary of Safety Related Changes from ROS 2 to Apex.OS

ROS 2

- Not real-time/deterministic
- No formal requirements compliant to ISO 26262
- No safety analysis
- No Static Code Analysis (SCA) or code coverage

Getting full MC/DC coverage and removing runtime memory allocations was challenging and took most of the time!

Apex.OS

- Several changes to improve real-time/determinism. Removed all runtime memory allocations and blocking calls.
- Formal requirements written and traced to design and test.
- SW FMEA carried on every package to derive additional requirements and/or restrictions.
- Full compliance to AUTOSAR cpp14 V3.19 coding guidelines.
- Full MC/DC coverage.

Thanks

[Apex.AI]
[Autoware.Auto]
[Autoware Foundation]
Contact: dejan@apex.ai