A Safety-Certified Automotive SDK to Enable Software-Defined Vehicles

Jan Becker*

Zusammenfassung: Automated driving, connected vehicles, e-mobility, shared mobility–all mobility disruptors rely on software but lack a unified software platform is preventing cross-domain software development. In the meantime, the vehicle compute and network architectures are moving to centralized high-performance computers, but the software implementation lags behind the hardware architecture. We have introduced Apex.OS, the first mobility software platform that can truly integrate across all in-vehicle domains. A primary vehicle operating system, robust and flexible enough to cover major systems throughout the vehicle and the cloud, enables user-focused development, just like iOS and Android SDK (software development kit) software development kit do so for embedded devices. This paper describes our development of an automotive SDK capable of covering all automotive software domains and already certified to ISO 26262 ASIL D. It addresses software development of driver assistance systems, automated driving, other safety-critical automotive systems, as well as applications for smart machines and IoT. **Schlüsselwörter:** vehicle OS, SDK, Apex.OS, vehicle software architecture.

1 Introduction

The automotive industry is moving toward new generations of the E/E-architecture based on centralized high-performance computers to support the megatrends towards electric, self-driving, shared, and connected vehicles. But the vehicle software architecture is lagging behind the hardware architecture. Today, automotive OEMs are stitching together disconnected software components to build proprietary platforms [8]. In the past, in a distributed E/E-Architecture, distributed and independently functioning electronic control units (ECUs) communicate with each other over CAN bus-based communication. For today's architecture, a central gateway was introduced to enable some cross-domain communication. Currently under development is the fourth generation of the E/E-architecture, which will see the introduction of domain controllers. These are computing platforms centralizing computing resources within one domain, such as the powertrain, chassis control, in-vehicle-infotainment, or ADAS and vehicle automation. This enables cost optimization, e.g., through consolidation of software and reduction of wiring harness cost but requires much higher bandwidth network communication and increases the requirements for safety and security. The fifth generation, currently being developed, will then see the introduction of centralized computing infrastructure, in which all major functionality is implemented on one high-performance computing platform, satellite sensors

*Jan Becker is CEO and Co-Founder of Apex.AI, Inc., 844 E Charleston Rd, Palo Alto, CA 94303, USA (e-mail: jan@apex.ai) as well as Lecturer at Stanford University.

are connected via a high-speed ethernet network and separation of domains is implemented in software. Overall, the architecture is rapidly moving from a complex system architecture with simple software to a simple system architecture with complex software. If done incorrectly, this will pose incredible challenges and risks, such as exploding software complexity, increased cybersecurity risks, incoherent cross-domain implementation, diverging non-consistent tooling, and so on. If done properly, this opens the opportunity to introduce modern software development processes and paradigms, a clean and scalable software architecture, cross-domain consistency and tooling, limited controllable cybersecurity risks, over-the-air update-ability, just to name a few.

2 How to get to software-defined vehicles?

The challenges of bringing about the Software-Defined Vehicle (SDV) are manifold and are resulting in widespread changes rippling throughout the automotive industry. The Software-Defined Vehicle is changing technology and toolchains. It is causing changes to the V-cycle development model's execution, openness, and iteration speed. It is changing organizational structures within companies. And it is changing the nature of OEM - supplier relationships.

Let's take a look at commercialization first. Automakers started by selling vehicles, then parts and accessories became an additional source of income after the initial vehicle purchase. Then came financing of sales and leases. Building Internet connectivity into vehicles opened additional potential revenue streams and kicked off the automakers' efforts to build their digital own ecosystems and services with the vehicle in the center of these ecosystems–but the world evolved differently. People still order their coffee through an app on their phone and not through the onboard UI and manufacturer ecosystem. The digital ecosystem and services are in the center and extend through apps to various devices at the edge. The vehicle has become one of many devices in these ecosystems, such as the phone, the pad, the home, health devices, etc, and services are consistently offered on just the few most successful digital ecosystems such as Apple or Android.

Automakers are determined to get there with large internal reorganizations and tremendous investments in their software abilities. *Software as product differentiator* is one of the factors driving OEMs to Software-Defined Vehicle and bringing more of their software development in-house.

3 Software-defined phones–what did we learn?

The market for operating systems is typically a winner-takes-all market. The industry segment does not matter, and that is why the automotive industry can learn a lot from what the mobile phone industry went through in the mid-2000s. Until then, a mobile phone's software was loaded onto the device at the end of the production line. The software was only updated if a significant bug was identified.

The introduction of the iPhone in 2007 popularized the separation of applications from the base software and hardware, turning the mobile phone into a software-defined device. It became a general-purpose platform upon which manufacturers could create an operating system with developer SDK and deliver updates over-the-air (OTA), enabling

developers to develop and deliver new applications. This drove the exponential growth of the smartphone market and compute performance of these devices. The two winners were Apple's iOS, which was first-to-market with superior user experience, and Android, which has grown to a 70% market share.

Previous platforms were swept aside. What led to the dominance of Android? Android is an open architecture with an SDK, is customizable and brandable, which saves the OEMs development cost and provides a rich application ecosystem that no single manufacturer (besides Apple) can achieve on their own.

4 Why a common OS and developer SDK?

Why is a common operating system (OS) with a software development kit (SDK) for mobility platforms relevant? Why shouldn't every manufacturer develop their own? Vehicles are becoming devices in a larger mobility ecosystem and are thereby competing with the larger mobile device market. Ecosystems derive value primarily through their size, so the larger an ecosystem, the more value it contains. Software developers and suppliers need to focus, so they are drawn to the most significant markets.

need graphic

Nevertheless, many automotive manufacturers started in-house initiatives to develop their own operating systems. Their goals are ambitious; various companies want to integrate their OS systems into their products as early as 2025. However, the time frame seems much too tight. Going into production in just three years with functions based on the operating system developed in-house is overambitious, especially for companies that are not software companies. Last but not least, the safety and security certification of the new software already takes up to three years. But what if the basis for such an operating system with developer SDK already existed? Could that speed development?

5 An approach to a common OS and developer SDK

In this paper, we aim to describe a world as well as a path to this world in which software throughout the whole vehicle is truly integrated end-to-end. A primary vehicle operating system on the outside–but really an SDK (software development kit) in the sense of Android SDK or iOS SDK on the inside–to abstract complexity away from the developers, provide common and open APIs as well as robust and flexible enough to cover major systems throughout the vehicle. What is needed to get to software-defined vehicles? We argue that the following requirements need to be fulfilled:

1. A standardized software architecture with open APIs to enable mutually compatible solutions ideally across many manufacturers, suppliers, and academia.

2. An awesome developer experience to enable developer productivity–based on the understanding that the quality of the developer experience is directly related to their productivity.

3. A software architecture that scales to massive software systems.

4. A software implementation based on modern software engineering practices.

5. Abstraction of the complexity of all underlying hardware and software.

6. All of the above with deterministic, real-time execution, and with automotive functional safety certification.

We draw an analogy to the field of robotics, which encountered similar challenges (with some exceptions w.r.t determinism and real-time) ten years ago and addressed the first five challenges by developing ROS (the Robot Operating System) as an open-source software framework. The authors are not aware of a single automotive company that is not or has not been using components from ROS or its rich software ecosystem in their development. But the first generation of ROS had its significant deficiencies. This has now changed: In 2018, the 2nd generation ROS 2 was released with improvements based on the lessons learned of using ROS 1 in hundreds of robotic and automotive prototypes over ten years. ROS 2 is based on a proven architecture that enables scaling to complex software systems. It is also based on a proven and standardized middleware instead of a homegrown data communication approach. In addition, with the release of Apex.OS (Apex.AI's proprietary fork of ROS 2), we addressed real-time and deterministic execution by rewriting the underlying implementation based on the same open APIs. And we developed a process to take this implementation through functional safety certification in record time. Here, we will describe the advantages of working with open APIs and an architecture that has been proven in use by tens of thousands of developers and in thousands of robots and automated vehicles. We will close by outlining a software development process that allows taking open-source software through functional safety certification into production vehicles.

6 Lessons Learned from Robotics

A robotics framework is a collection of software tools, libraries, conventions, and APIs made to simplify the task of developing software for a complex multi-sensor, multi-actuator system, like a vehicle. A well-implemented robotics framework is implemented with the separation of concerns in mind. These concerns are introduced in the paper by Vanthienen as 5Cs [14]: configuration, coordination, composition, communication, and computation. Writing the software this way allows for maximum reusability, testability, reliability, portability, maintainability, extensibility of components. In most cases, the use of a robotics framework dictates the general architectural principles of the developed software. For example, if the software is centralized or decentralized, real-time or non-real-time, etc. A key component is the middleware, which is the glue that holds together the numerous components of a robotics framework. The most basic task of the middleware is to provide the communications infrastructure between software nodes in an autonomous vehicle. The typical use case for a robotics framework is to provide the essential interfaces between high-level (software) and low-level (hardware) components of the system. These interfaces and components consist of various operating system (OS) specific drivers that would take a single developer a significant amount of time to develop. The software framework should abstract away ECUs, OSes, physical transport layers, and interface to the components needed for offline (e.g., mapping, simulation, testing) as well as online operations (user applications).

6.1 ROS 1

Various efforts at Stanford University in the mid-2000s involving integrative, embodied AI, such as the STanford AI Robot (STAIR) and the Personal Robots (PR) program, created in-house prototypes of flexible, dynamic software systems intended for robotics use–which they named ROS[10]. In 2007, Willow Garage, a nearby visionary robotics incubator, provided significant resources to extend these concepts much further and create well-tested implementations. The effort was boosted by countless researchers who contributed their time and expertise to both the core ROS ideas and to its fundamental software packages. Throughout, the software was developed in the open using the permissive BSD open-source license and gradually has become a widely used platform in the robotics research community.

From the start, ROS was developed at multiple institutions and for multiple robots, including many institutions that received PR2 robots from Willow Garage. Although it would have been far simpler for all contributors to place their code on the same servers, over the years, the "federated" model has emerged as one of the great strengths of the ROS ecosystem. Any group can start their own ROS code repository on their own servers, and they maintain full ownership and control of it. They don't need anyone's permission. If they choose to make their repository publicly available, they can receive the recognition and credit they deserve for their achievements and benefit from specific technical feedback and improvements like all open-source software projects.

The ROS ecosystem now consists of tens of thousands of users worldwide, working in domains ranging from tabletop hobby projects to vehicles to large industrial automation systems. Despite its large success, ROS 1 did not find its path to automotive production programs because of the following reasons: low code quality, non-standard, non-automotive middleware, lack of real-time capabilities, no nodes with managed lifecycle, no security, lack of testing and documentation, no support for automotive ECUs.

6.2 ROS 2

To address the aforementioned shortcomings, the Open Source Robotics Foundation (OSRF) started a new project in 2013, ROS 2. Open Robotics considered autonomous vehicles as well as the following use cases before starting development for ROS 2:

- Small, embedded platforms: ROS 2 has a smaller and more optimized codebase compared to ROS 1, which can run on aarch64 computer architectures.
- Real-time systems: ROS 2 communication mechanism, memory management, and threading model allow for it to be real-time upon hardening.
- Production environments: Coupled with real-time constructs, ROS 2 can run on an RTOS, such as QNX.
- Prescribed patterns for building and structuring systems: While ROS 2 has the underlying flexibility that is the hallmark of ROS 1, ROS 2 provides patterns and supporting tools for features such as life cycle management and static configurations for deployment.

Now let's have a look at how ROS 2 fulfills the promises given in the introduction of this article.

7 A standardized software architecture with open APIs

Developing a standardized software architecture that can satisfy applications ranging from an embedded microcontroller running on a drone or radar sensor to the centralized high-performance ECU for the autonomous car is no easy task. Many different aspects such as a hardware abstraction layer, an OS abstraction layer, a runtime layer, support for various programming languages, non-functional performance, security, safety, software updates, tools for the development, debugging, recording and replay, visualization, simulation, tools for the continuous integration and continuous deployment, interfaces to the legacy systems (such as AUTOSAR Classic), execution management in the user applications, time synchronization, support for hardware acceleration, model-based development, and more have to be considered. To integrate these aspects into one coherent framework, the following design decisions were followed in the design of ROS 2:

- Adopt positive lessons learned from the extensive deployment of ROS 1
- Apply negative lessons learned from the extensive deployment of ROS 1
- Use hourglass design to keep the core of ROS 2 as a single code base
- Use industry-proven middleware and focus on the layers above the middleware
- Newly developed mechanisms for security, safety, software updates, interfaces to the legacy systems (such as AUTOSAR Classic), execution management in the user applications, time synchronization, support for hardware acceleration, model-based development

With these aspects in mind, a new architecture shown was created for ROS 2. The largest change from ROS 1 to ROS 2 is that the underlying communication layer was implemented in a plug-in-like interface for many types of middleware solutions and also provides tier 1 support for DDS implementations. On top of the middleware layer is a middleware abstraction layer called the ROS middleware API, which is used to prevent a DDS vendor lock-in. Implementation of the middleware layer for example, SOME/IP is with this design rather trivial. On top of this layer are the ROS 2 client libraries written in C++ and Python. Using these client libraries, developers can write application code. With this so-called hourglass design, the thin client libraries always wrap around the same code base for operating system and communication.

One of the concepts that proved itself the most in ROS 1 was the concept of a node as a central logical unit. The node component brings together different publisher and subscriber patterns, threading, queues, and the execution model–all abstracted behind the intuitive set of APIs. A node can subscribe to one or more data streams (called topics), the data is then aggregated and processed in a single-threaded or parallel loop, and the result is published on another topic. The node can also be parametrized, composed into one or more processes, and it also comes with its own lifecycle for starting and stopping. Nodes also emit diagnostics information such as logging, heartbeat, and node state information.

One of the missing concepts in ROS 1 was the concept of executor, which was added in ROS 2. The executor provides spin functions and then coordinates the nodes and

callback groups by looking for available work and completing it, based on the threading or concurrency scheme provided by the subclass implementation. An example of available work is executing a subscription callback or a timer callback. The executor structure allows for a decoupling of the communication graph and the execution model–a feature much needed when building real-time and deterministic systems.

ROS 1 provided standardized messages [1] for diagnostics, geometric primitives, robot navigation, and common sensors such as laser range finders, cameras, and point clouds. This allows for having the agreed-upon contract between different software and hardware modules and in turn allows for great flexibility and reusability of developed software components. These messages were also adopted in ROS 2 but implemented with the OMG IDL language, which enables e.g., message versioning and message forward and backward compatibility.

ROS 1 had very high non-functional performance characteristics, especially in terms of CPU usage and high bandwidth which allowed the developers to cope with the throughputs of up to 6 Gb/s of data as produced by for example an autonomous driving car. In ROS 2 this had to be preserved and even further enhanced by the introduction of e.g., zero-copy transport mechanisms such as iceoryx [2]. These are just a few examples of proven in the field which could be improved because ROS 1 is an open-source framework [11] and because there is a large community of users constantly running into the edge cases and the best software is developed through the coding iterations and rather than through implementation solely based on specifications.

7.1 An awesome developer experience

ROS community has developed a various set of desktop and cloud tools that allow the users to introspect and debug their systems, record, replay, and visualize the data, and emulate and simulate their systems. For the full set of tools, we would like to direct the user to the dedicated webpage [13]. For ROS 2, we would like to single out two particular tools that contributed to the improved developer experience and productivity.

The ADE Development Environment [1] is a modular Docker-based tool to ensure that all developers in a project have a common, consistent development environment. ADE creates a Docker container from a base image and mounts additional read-only volumes in /opt. The base image provides development tools (e.g., vim, udpreplay, etc.), and the volumes provide additional development tools (e.g., IDEs, large third-party libraries) or released software versions. Furthermore, ADE enables easy switching between versions of the images and volumes.

Arguably the most significant effort required for the production deployment of autonomous vehicles (AVs) is the proof that the system is safe for homologation. To provide such proof, a holistic, software-first, approach to testing for autonomous vehicles must be available. In other words, we need a framework that enables writing of the relevant tests and supports extracting the proofs. To develop such a framework, it is important to look at the automotive V-model and to understand what kind of tests one is expected to write: Unit tests, component tests, integration tests, sub-system tests, system tests, non-functional tests, acceptance tests. Furthermore, it is important to understand how input data is generated for these tests: Manually created, manually generated, simulation generated, vehicle recordings, and in what kind of environments the tests are executed: Software-in-the-loop (SIL), Hardware-in-the-loop (HIL), Vehicle-in-the-loop (VIL). Lastly,

it is important to understand what kind of key performance indicators are to be extracted from the tests.

There are plenty of tools available for unit and component testing (e.g., [4]) but upon extensive research, we were unable to find a framework that would satisfy other types of tests, cover all other input data and environments and that would clearly between the following phases: test configuration, test setup, test running and results logging and extraction. We thus wrote a tool in the open source called launch testing [5], which is a framework for the integration, system and sub-system, non-functional as well as acceptance testing of applications written with ROS 2 and Apex.OS.

7.2 A software architecture that scales to massive systems

By "Software architecture that scales" we refer to scaling with the number of developers, scaling in terms of the non-functional performance, scaling in terms of the integration of third-party automotive components, and scaling in terms of centralized and decentralized applications. ROS (and thereby Apex.OS) is the most field-proven framework worldwide in terms of user adoption. Per the last ROS community metric report citeROS-metrics, there are more than 200.000 known users of ROS worldwide, and there are presumably significantly more unknown developers in corporate development. Key enablers for this are open APIs, an ecosystem of tools, best development practices, and tight integration with various CI/CD systems.

The ROS and Apex.OS non-functional performance has been described in detail [9] with respect to the efficiency of abstraction layers and the efficiency of the transport protocols resulting in the acceptable CPU utilization, memory consumption, latency, page faults, context switches, and other such system metrics. As explained in [10], it is also important to have a properly designed tool for unbiased measurements and a methodology that mimics that of the actual application.

7.3 A software implementation based on modern software engineering practices

To have a robust and maintainable codebase one needs to use a) well-defined but friendly software development process and b) state-of-the-art software development tools. Such SW development process encompasses all the steps from the input requirements all the way to the CI/CD. The keys to the success of such a development process are:

- An integrated development environment, which at Apex.AI is centered around Gitlab, Gitlab CI and docker.
- An integrated IDE, which at Apex.AI is Clion. Clion provides all of the state-of-the-art features such as code completion, debugging but also integration of external tools such as e.g., gtest, valgrind, different build tools, doxygen, tool for code test coverage.
- The steps implementing the development process must be fast to allow quick coding iterations.

- The local development environment and the CI/CD must be equivalent to be able to reproduce CI failures.
- Integrations with the 3rd party tools, such as the JAMA requirements management tool, must be custom tailored to the particular team to allow for the quick fixes and extensions.
- The main code repository should be as monolithic as possible and all of the development artifacts (design documents, code, tests, documentation, etc) should be as co-located as possible.

8 From ROS 2 to Apex.OS Cert–ISO 26262 [5] ASIL D certification of an open-source codebase

Apex.OS Cert is based on our fork of ROS 2 Dashing. Starting with a technical safety concept (TSC) for Apex.OS, which defined an assumed use case and ODD (operation design domain). For the first iteration of Apex.OS Cert assumed use case consisted of a rather simple scenario where one or more publishers communicate data to one or more subscribers. Of the 5Cs [14] of a typical robotics framework (communication, coordination, computation, composition, and configuration), hazards derived were restricted to data communication. This reduced the scope of Apex.OS Cert safety concept but at the same time was sufficient to meet customer requirements at the time. The next step was to determine the boundary of Apex.OS Cert. We decided to restrict the boundary to the source code and to add libraries to the scope in future releases of Apex.OS Cert. At this point, we wrote an initial safety case of Apex.OS Cert. The Apex.OS Cert safety case is an internal document that lays out what are our safety objectives and what artifacts are generated. An safety audit resulted in tacit approval for both the aforementioned TSC and safety case. While defining the safety case, we also inventoried all the development tools that were required to run TCL (Tool Confidence Level) reports (ISO 26262-8:2018 [6]) and planned our processes in FSLC (functional safety lifecycle) such as our change management, safety culture, code review enforcements and so on. These plans were also captured in functional safety management (FSM) artifacts as dictated in ISO 26262-2:2018.

9 Challenges

Several technical challenges were encountered during this process, including the following. Achieving 100% structural coverage: ROS 2 packages came with low MC/DC coverage, and we selected a modern and certified coverage tool to help us achieve complete coverage. ROS 2 is written in modern C++ with heavily templated constructs. Available coverage tools failed to process complex C++ constructs such as back-to-back lambda functions. We worked with our tool vendors to identify solutions. Furthermore, defensive coding methodologies can make it difficult to reach conditional statements that are gated by external libraries over which we have little control. We reverted to GoogleMock [3] techniques in such cases. Checking for runtime memory allocations: Applicable tools for checking runtime memory allocations and blocking calls are not available to our

knowledge. We repurposed LTTng [7], the Linux tracing tool next gen, to accomplish this task. Automation was difficult, and issues such as instrumenting code had to be done manually, which is undesirable for obvious reasons. Making Apex.OS runtime execution deterministic: ROS 2 uses standard containers which allocate memory explicitly. We reverted to our own containers for string, vector, and map/set to not only make Apex.OS Cert runtime memory static but also make the execution as deterministic as possible. Development tool classification and qualifications: ROS 2 uses external development tools for activities such as parsing/configuring (e.g., YAML) and code generation (e.g., rosidl_generator_cpp). TCL reporting led to TCL2 (medium confidence) and consequently the tools had to be qualified. Qualification of external tools is challenging since access to source code and test base is often not available. Qualifying a large open-source tool such as YAML parser (that also includes a library) is prohibitively resource-intensive and time-consuming. We reverted to proprietary solutions for each purpose.

10 Summary and Outlook

We have been able to show that certifying an open-source project to the highest level of integrity is possible with the right expertise and strategic choices of scope and tooling. Apex.OS Cert provides manufacturers and suppliers a fast track and scalable process to build a safe automotive software stack. Our customers have shown applicability to a wide range of automotive applications not limited to autonomous driving. We have been able to accomplish development and certification in a very short amount of time based on our prior expertise in ROS and through reuse of a proven architecture and tooling. For future work, Apex.AI will not only expand the technical safety concept of Apex.OS but also certify libraries including the important transport layer (middleware) libraries. Apex.OS is also in the process of being integrated with AUTOSAR Adaptive and SOME/IP.

References

[1] *ADE environment*, `https://ade-cli.readthedocs.io/en/latest/`, last accessed 2021/05/08.

[2] *Eclipse iceoryx*, `https://github.com/eclipse-iceoryx/iceoryx`, last accessed 2021/05/08.

[3] *GoogleTest Mocking (gMock) Framework*, `https://github.com/google/googletest/tree/master/googlemock`, last accessed 2021/05/08.

[4] *GoogleTest Framework*, `https://github.com/google/googletest`, last accessed 2021/05/08.

[5] *ISO 2626 Road vehicles–Functional safety*, `https://www.iso.org/obp/ui/#iso:std:iso:26262:-9:ed-2:v1:en`, last accessed 2021/05/08.

[6] *ROS launch-testing*, `https://index.ros.org/p/launch_testing/`, last accessed 2021/05/08.

[7] *LTTng tracing framework for Linux*, https://lttng.org/, last accessed 2021/05/08.

[8] McKinsey & Company, *Mapping the automotive software and electronics landscape through 2030*, https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/mapping-the-automotive-software-and-electronics-landscape-through-2030, last accessed 2021/05/08.

[9] Pemmaiah, A., Pangercic, D., Aggarwal, D., Neumann, K., Marcey, K., *Performance Testing in ROS 2*, *https://www.apex.ai/post/performance-testing-in-ros-2*, last accessed 2021/05/08.

[10] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A., *ROS an open source operating system*, In: ICRA workshop on open source software 2009, http://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.

[11] *ROS Community Metrics Report*, http://download.ros.org/downloads/metrics/metrics-report-2020-07.pdf, last accessed 2021/05/08.

[12] *ROS core stacks*, https://github.com/ros, last accessed 2021/05/08.

[13] *ROS tools*, http://wiki.ros.org/Tools, last accessed 2021/05/08.

[14] Vanthienen, D., Klotzbücher, M., Bruyninckx H. *The 5C-based architectural Composition Pattern: lessons learned from redeveloping the iTaSC framework for constraint-based robot programming.* In: 9th Journal of Software Engineering for Robotics, pp. 17-35. https://lirias.kuleuven.be/1748457.