# Open standards enable continuous software development in the automotive industry

Markus Glaser, Charles Macfarlane, Benjamin May, Sven Fleck, Lukas Sommer, Jann-Eve Stavesand, Christian Weber, Duong-Van Nguyen, Enda Ward, Illya Rudkin, Stefan Milz, Rainer Oder, Frank Böhm, Benedikt Schonlau, Oliver Hupfeld, Andreas Koch

**Abstract**—Until recently, the automotive industry was primarily focused on design, development of electronics and mechanics, and manufacturing. Nowadays the software component of new vehicles has become a large portion of the development cost, driven by adding numerous new sensors, intelligent algorithms, very powerful and specialized processors and a highly complex user experience to the vehicle. In addition, requirements for high-performance and real-time processing as well as the vehicle's distributed architecture bring challenges, but moreover supply chains further complicate development. In this article a high-level overview of vehicle development is provided, followed by a deep dive in the different software development processes, languages and tools that are required for efficient development of the next generation of intelligent vehicles. This paper especially explores **SYCL™**, an open standard from The Khronos™ Group for high-performance programming of heterogeneous multicore processor system.

**Index Terms**—Automotive, continuous development, open standards, SYCL, CUDA, AI driven, software driven, ADAS, heterogeneous computing, AI, Machine Learning

## 1 Introduction

Modern cars utilize 100+ ECUs and reached 100M lines of code by 2015 [1], [2]. Code complexity is expected to increase exponentially, resulting in over 650M lines of code by 2025 [3] for every vehicle and over 1bn lines to achieve a Level 5 autonomous vehicle.

Development cycles are getting shorter with Continuous Integration (CI) and Continuous Delivery (CD) being established in the automotive industry, and these practices are explained and explored further in this paper. Technology today allows for the latest updates automatically, instantly and "free", helping to improve and extend functionality. This development began with agile software management methods, gaining popularity in application software development, enabling fast release-cycles. Software developers can provide a new release to a broad base of customers at the push of a button. Car companies are now focusing more on software and features, bringing them more in line with techniques used today to delivery software applications to always connected devices.

However, "Fail Fast" has become the driving motive of innovation-generating software companies. Within automotive, any bugs in software can have unexpected and costly results. Assuming a car manufacturer intends to "Fail Fast", they will have to do so before they deliver the new software, performing thorough checks of their software regularly if they intend to deliver continuously.

Another demand from the automotive industry is WP.29 within the UN World Forum, which includes a requirement that software must be continuously maintained for the lifetime of the vehicle.

Three main challenges are covered in this article: The software development processes, the shift towards CD / CI, and the programming of future - heterogeneous systems using open standards (e.g. SYCL). To this end an overview of the automotive landscape is given, where functional safety standards are now the norm and where open standards for software are becoming the solution for the automotive industry, achieving the demands of ADAS developers and overcoming software development challenges.

## 2 Status quo

### 2.1 Digital transformation

Tesla®, a Silicon Valley company, is the only automobile manufacturer credited with achieving CD [4], [5] today. The traditional automotive industry is facing a radical and fundamental transformation [6]. Besides electrification of the powertrain, new vehicle development projects are rapidly adopting a broad range of new sensing and AI technologies. The overall goal is to provide a more comfortable and, ideally, automated ride for the driver and passengers, while reducing accidents and increasing safety. The intelligent sensing systems that support vehicle operation are often referred to as Advanced Driver-Assistance Systems (ADAS). We will use the ADAS/AD term broadly in this paper; to describe both systems that provide either fully autonomous driving capabilities (AD) and systems that assist the driver in operating the vehicle (ADAS).

These new ADAS solutions are driving a major transformation in the vehicle development process. Key elements are depicted in Fig. 1.

Within this paradigm shift two major components come together:

1) Software is becoming the key value proposition. It is a major transformation and requires the corresponding infrastructure and processes to enable

| Traditional Approach | digital transformation | Future Approach |
|---|---|---|
| hardware | | software, data, AI |
| proprietary | | (open) standards |
| static, frozen | | field upgradable (OTA) |
| single development | | continous development (CD) |
| single integration | | continuous integration (CI) |
| single platform usability | | cross platform usability |
| software tied to hardware | | decoupling of software and hardware |
| tied to project (hw, Tier1s, Tier2s) | | robust against changes (hw/sw components, Tier1s, Tier2s) |

Fig. 1. High level overview of the automotive digital transformation.

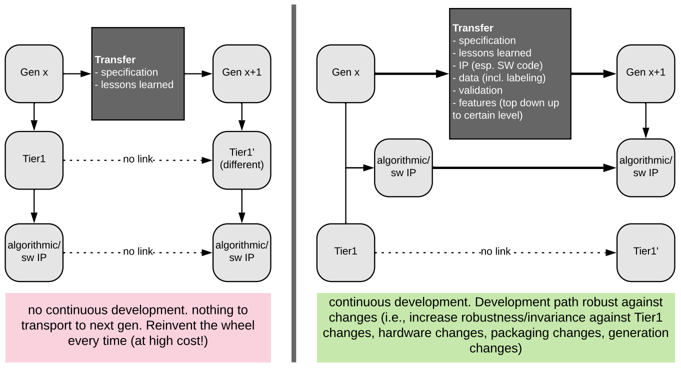

Gen x
Transfer
- specification
- lessons learned
Gen x+1
Tier1
no link
Tier1' (different)
algorithmic/ sw IP
no link
algorithmic/ sw IP
no continuous development. nothing to transport to next gen. Reinvent the wheel every time (at high cost!)
Gen x
Transfer
- specification
- lessons learned
- IP (esp. SW code)
- data (incl. labeling)
- validation
- features (top down up to certain level)
Gen x+1
algorithmic/ sw IP
algorithmic/ sw IP
Tier1
no link
Tier1'
continuous development. Development path robust against changes (i.e., increase robustness/invariance against Tier1 changes, hardware changes, packaging changes, generation changes)

Fig. 2. **Left:** Traditional software development organization, with no software IP being preserved between generations due to tight coupling to proprietary hardware and tool interfaces.. **Right:** Benefits of open standards in a continuous software development process. Due to standardization of hardware and tool interfaces, software IP can be preserved and development cost can be reduced.

its full potential (CD of software, software over-the-air (OTA) upgrades, industry defined software standards, software decoupling from hardware).

2) Data captured, labeled, and processed/analyzed is becoming a major asset. AI based perception systems significantly outperform classical, hand-engineered software algorithms, especially for AI based software. Data is crucial for the digital transformation.

The changes are industry-wide and not only affect technology development, but also business partnerships, business models, and even the value proposition to the consumer. One example is the recently announced partnership between Mercedes® and Nvidia®. Nvidia, with its roots in consumer electronics and gaming, has evolved to be a leading supplier in AI and brought its technology into the automotive domain. Mercedes, with many other companies, stated their commitment toward a software-defined vehicle development approach and Nvidia are providing a highly programmable solution for this. MobilEye® has similarly created an intelligent platform and achieved success as an early solution bringing AI processing into the vehicle.

In this paper, we give an overview of the underlying unique requirements of automotive ADAS, with a focus on software development. We identify relevant key performance indicators (KPIs) and review and assess multiple approaches.

## 2.2 Automotive software development process

The automotive digital transformation in general, and the advent of ADAS in particular, increases the demand for software dramatically, resulting in unprecedented challenges with regard to organization of the software development process and the interaction of OEMs and suppliers.

### 2.2.1 *Organizational transitions*

Until recently, ADAS development projects were mainly treated as isolated activities, driven by the OEM with its awarded respective Tier 1 suppliers and the underlying Tier 2 ecosystem. The suppliers delivered hardware units to car manufacturers with all the necessary software pre-specified in terms of thousands of requirements. Now, these requirements are no longer static, but are changing even after delivery of the car. Complex ECUs include software from multiple vendors, which triggers the need for fast software integration that remains safe and provides high quality.

The continuously changing requirements, and the fact that software IP will be an important asset for automotive companies in the future, means a change towards a CD process is inevitable. In Fig. 2, the left side summarizes the traditional approach where the software IP is tightly coupled to the Tier 1 supplier's proprietary tools and hardware, and software IP cannot be preserved between different generations of vehicles, resulting in repeated high development costs.

In Fig. 2, the right side illustrates the benefits when open standards are used to develop the algorithms and software. As the interfaces to hardware and tools are specified by open standards, software IP can be preserved and benefit the enhancements in newer vehicle generations, even when switching Tier 1 suppliers and underlying hardware. This reduces development cost, time to market and preserves important company assets.

Due to the high volume of automobiles, the development effort (and cost) is less crucial compared to the bill of material (BOM) of such systems. This forced traditional OEMs to squeeze every bit out of a system by using proprietary approaches, hardware, and tools. This is in sync with the OEMs traditional key competence of performing **integration**, whereas the underlying Tier 1 and its Tier 2 partners focus on **innovation**. We see that **standards enable the automation of integration**. As many functions that have previously been reserved for high-class vehicles only now become mandatory even in lower end (high volume) vehicles, the burden of reinventing the wheel with significant innovations and risks becomes a new factor. This is true for both surround and in-cabin sensing (driver monitoring), driven e.g., by Euro NCAP requirements.

### 2.2.2 *Software development process*

So far, the main focus of the software development process in the automotive domain was the short time window to develop until start of production (SOP) for the lead vehicle. After initial SOP, a rollout phase started where the main task was to integrate this resulting system in additional upcoming production lines and variants. Once the car was in customer's hands, there was neither the focus, nor the technical possibility to update certain features for bug fixing or adding innovations. While remote updates OTA are commonplace, very few cars can do this today. Moreover, legacy vehicle architectures with their fragmented ECU topology do not allow for OTA updates, and the appropriate cyber security measures required are missing, as is any vehicle communication capability.

Installing new software is technically feasible in workshops, however the development resources from the corresponding Tier 1s are usually already focusing on the next project/generation for a potentially different program or OEM (except for critical or legislative bug fixing).

Software driven companies show a different approach where CD is a fundamental ingredient. Tesla vehicles for instance all support over-the-air system software updates that are installed without having to bring the car to a garage. This enables rapid and frequent software improvements to the whole vehicle that continuously enhance the operation and user experience. The update cycles of the consumer and smartphone devices have driven the demand for centralized software management and network connectivity for access to the cloud. This transition might continue towards a mixture of zonal and domain-oriented architectures, where partial updates based on functions or zones are becoming a service that will be distributed towards available devices in the vehicle itself, using information from onboard sensing and offboard information. A key challenge in automotive is that functionality needs to fulfill the expectations of a long vehicle lifetime, high quality, safety, and security. Software and system failures can create critical states which in the end can risk people's lives and cause severe damage. This might lead to a scenario where the best-of-both-worlds will coexist: the old traditional safety and quality-oriented heritage of automotive, combined with the new demand for rapidly evolving complex software. In Fig. 3 the ecosystem players' roles are classified relative to their awareness for continuous development (and thus their belonging to the "Future Approach" category on the right of Fig. 1). Emerging non-legacy companies obviously are not required to perform such a transformation but start in the green area from the beginning.

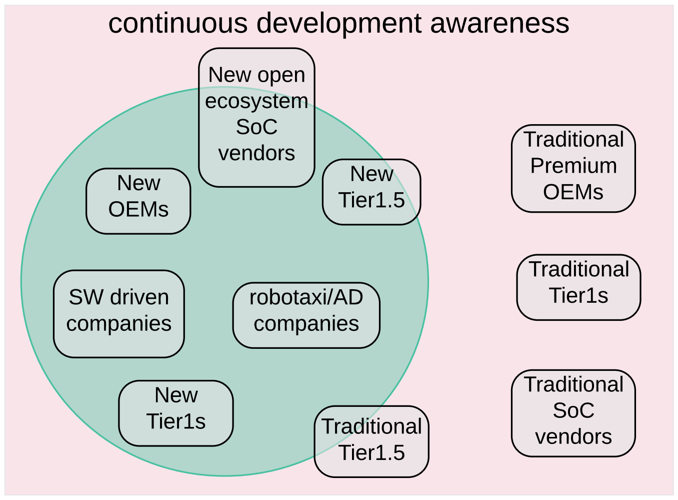


Fig. 3. Overview of old and new players of the automotive ecosystem w.r.t. continuous development awareness.

## 2.3 Distributed and heterogeneous systems

Radars, LiDARs, cameras, ultrasonic, GPS and mapping data – so many sources of data need to be processed, fused, and interpreted to generate a set of controls for the car to execute. When including the data checking, the fail-safe systems with redundancy and the compute requirements, the scale of complexity expands further than the current system can support. Also, from high-end vehicles down to economy versions with reduced systems, there needs to be a modular and flexible system architecture to cope with the market needs going forward. Sensors are distributed around the car and vary in features, performance, functionality, and interface protocols. Most sensors integrate analysis systems to reduce the volume of raw unprocessed data being sent around the vehicle. Processed and reduced data, such as images or 3D transformation coordinates, is used instead, so as not to overload the centralized network and compute system with all the processing. Most sensor units contain an integrated processing unit which itself will require updating, improving, and maintaining throughout the lifetime of the car. The processed data and alerts are then fed into a sensor fusion unit, which has to make sense of its inputs, construct an operational scene using new data and previous history to build a model of the type and intentions of objects in the vicinity, and then execute a safe course of action. "Intentions" is an extremely complex topic by itself. It is of no use just identifying an object and location, but it is also necessary to know what it is likely to do next. From the behavior of an animal based on its historic movements (pointing away or towards your path, fast or slow, disturbed or calm, behavior when frightened etc.), to people, vehicles, objects (shopping bag blowing in the wind or a solid object), they all need to be interpreted to determine which maneuver to propose to the car.

All these many different stages in the system have different processing requirements and need very different processor types, therefore demanding heterogeneous systems integrating many different processing architectures are now common for the implementation of ADAS functionalities.

The most regularly used and commonly understood ECU component is a CPU, e.g., as provided by ARM or RISC-V. They are good at sequentially processing simple pieces of data and irregular control functions. Many companies are capable of supporting CPUs into production. They are predictable and often programming can be done in assembly, C or more recently C++.

For real-time data processing with more complex algorithms, a Digital Signal Processor (DSP) is regularly used. This starts to cause an issue for programming, since most DSPs require the developer to understand their microarchitecture. This is the first barrier for a programmer alongside the issue for updating and maintaining the DSP functions throughout the lifetime of a vehicle.

Graphical Processing Units (GPUs) have been regularly used for providing graphics to the car dashboard and have been used for many years with a good understanding of how to achieve this reliably. While GPUs have been used in non-automotive products to provide vision and AI functions, thanks to their vector-like processing architecture, they have not yet been utilized much beyond the intended graphics functions in vehicles. However, the programming of these devices is popular thanks to the OpenGL$^{\text{TM}}$ (graphics) and OpenCL$^{\text{TM}}$(compute) open standard interfaces, and CUDA, a well established proprietary interface for Nvidia GPUs.

Another popular processor is the Vector Processing Unit (VPU), often at the heart of computer vision due to its ability to process an array of pixels in an image simultaneously. Like DSPs, the programming of these devices is not standardized and requires very special attention to the microarchitecture and assembly instructions to achieve an efficient algorithm implementation. Often, these processors are supplied as a black-box, i.e. system developers take what is delivered by the semiconductor company and are unable to provide their own functions or value-add.

The current generation of processors and accelerators is especially suited to computer vision, AI, and machine learning. These are the latest technologies that need to satisfy the processing needs of ADAS. They are very heterogenous, providing different accelerators and programming interfaces.

An example of such a heterogeneous system integrating a whole range of different processing units is Tesla's FSD chip [7]. It combines a 12-core CPU (divided into three clusters of four cores each), a GPU, an image-signal processor (ISP) and a specialized neural-network accelerator (NNA) in a single System-on-Chip (SoC).

Without open standards for programming models on different processor architectures, the interfaces and tools require the manufacturers to fully commit to one solution. This goes against the desire of every manufacturer to achieve CD, have a flexible supply chain and optimal solution per vehicle SKU.

For all of the processor flavors available, and the different applications within the ADAS pipeline, they all present one very common problem – how does a software developer program ADAS systems in a consistent way avoiding substantial re-writing for each generation, and how to keep updating them for the lifetime of the vehicle?

## 2.4 What are Continuous Integration and Continuous Development?

CI is the practice of regularly integrating and testing all the developers' working software into a shared code repository on a regular basis. The merging of the software includes any new features or improvements. As well as software, modifications to data sets, test units, configuration parameters or build scripts along with supporting documentation changes are also integrated. With agile software development being widely adopted, long integration cycles are becoming obsolete and even obstructive. In 1991, the term "Continuous Integration" was first used by Grady Booch [8] to describe an effective, iterative way of building software. The technique was quickly adopted into the set of techniques used in Extreme Programming and detailed guidelines were summed up by Fowler in 2006 [9].

An established CI process maximizes efficiency by minimizing the time it takes to build, test, and release new features, while ensuring that a quality standard is maintained.

CD describes a process for iterative software development and is an umbrella over several other processes including CI, continuous testing, continuous delivery and continuous deployment.

The CI cycle repeats on every new commit of a change to the software stack being developed. Immediately the CI pipeline starts code analysis, compilation, unit and integration tests in a sequence. An application can contain many layers. Each layer in the stack can contain many dependencies on libraries for functions such as math, scheduling or communication protocols. Any change to any one layer can trigger a new CI cycle. The CI pipeline ends with integration tests performed on the hardware, generally a single ECU in a vehicle in the field, otherwise known as Hardware-in-the-Loop or Open-Loop test benches.

Note the added challenge here. CI / CD in a pure software company is carried out (idealized) in one place on one server and so the turn around on knowing a very recent commit being successful can be realized in minutes or hours. For automotive however, the CI / CD is likely to be deployed and distributed across several physical systems in different locations. A further additional challenge when out in the field is the consumption and generation of test data and its transportation. A test vehicle can consume and produce petabytes of data, as sensors bandwidth can reach or exceed 40 GBit/s ($\approx$19 TB/h) [10], [11].

CD seeks to automate and streamline the process of building, testing and deploying new code into a live or staging environment.

What does CD promise?

- Faster delivery of new features
- Better quality product
- Risk avoidance
- Lower resource requirements
- Increased productivity

The main challenge for the automotive developer is that a car is considered a safety-critical system, which means that functional safety checks have to be performed before any deployment to the production environment. Typically, analysis methods like FMEA (Failure Mode and Effects Analysis) or STPA (System Theoretic Process Analysis) are used. These are used to carry out safety analysis identifying possible hazardous scenarios and testing them against the release candidate.

Since many parts in the ADAS AI compute stack are from multiple vendors, open standards provide an ideal solution. As for most of the numerous parts in a car, the supplier has a duty to notify the OEM of any improvements or changes resulting in an update release cycle.

## 2.5 Standardization

A development in parallel with ISO 26262, is the advance made by AUTOSAR in an effort to develop a global standard for system architecture, so that the basic software functions of automotive ECU's can be standardized. AUTOSAR (Automotive Open System Architecture) is a development

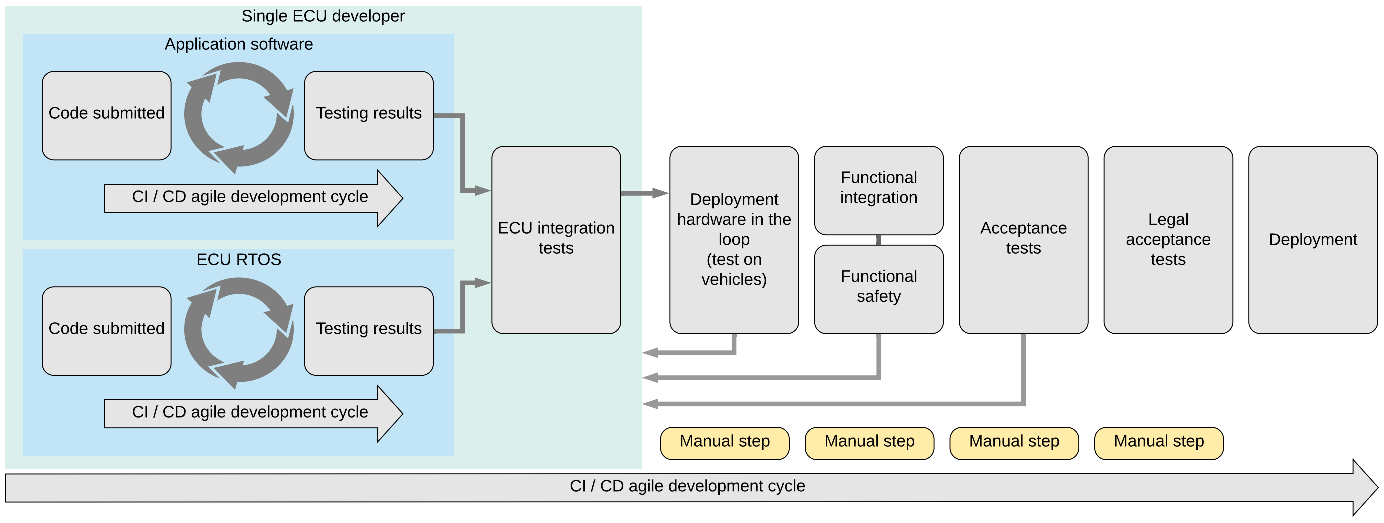


Fig. 4. Continuous development pipeline for automotive

partnership between leading OEM's and tier one suppliers in the automotive industry. The common objective is to create a development base for industry collaboration on basic functions while providing a platform which still encourages competition on innovative functions. The development partnership has been formed with the goals of:

- Standardization of basic software functionality of automotive ECUs
- Scalability to different vehicles and their platform variants
- Transferability of software
- Support of different functional domains
- Definition of an open architecture
- Collaboration between various partners
- Development of highly dependable systems
- Support of applicable automotive international standards and state-of-the-art technologies.

Mastering the ever-increasing complexity of future ADAS requires abstraction and **standardized, open interfaces** to enable robustness against changes, especially those made to individual modules, components, Tier 1s, Tier 2s, target car lines with different combinations of adjacent systems of multiple generations (”fragmentation”) or even system generations themselves.

Standards are thus crucial to enable this endeavor.

### 2.5.1 *Old standards need to be updated*

ISO 26262, the automotive safety standard derived from the IEC61508 was released in 2011, driven by tragic vehicle accidents caused by software failures. This includes Toyota's 2005 incident where a faulty electronic throttle control system is alleged to have caused unintended acceleration. ISO 26262 was published to provide a framework that enables the identification of potential risks of software and hardware failure, and to apply a standard to ensure the functional safety of automotive electronic and electrical systems. The standard can be used for all activities within the development life cycle of safety-related systems including:

- Management, development, production, operation, service and decommissioning
- The outlining of a risk-based approach (through Automotive Safety Integrity Levels - ASIL)
- Identification of unreasonable residual risk
- The validation and confirmation of safety goals
- The management of requirements for with suppliers

ISO 26262 has been commonplace in the European, American, Japanese and Korean markets for many years. The adherence by some automotive Tiers to old code quality standards stifles the development of applications for vehicles today for the convenience of obtaining a recognized baseline level of quality. The ISO 26262 standard does not mandate any particular quality standard for software development. It asks that requirements for software be followed according to the level of functional criticality which is determined by the risk of injury to vehicle occupants. Functional safety standards when released lag behind the state-of-the-art and the second edition ISO 26262:2018 is no exception. It presents a difficult balance between an established and welcome baseline to work from with that of todays‘ emerging complex ADAS solutions. The automotive industry has recognized these gaps in safety standards such as:

- Cyber-security
- Programming parallelism and indetermistic behavior
- Machine learning and training of AI systems
- Human interaction with autonomous systems

There are many institutes and groups producing standards to address these gaps and show the public and government regulators that they are self-regulating and managing the current and future concerns for safety in autonomous systems. Some of these standards are listed below.

Established quality standards:

- AUTOSAR (Adaptive) C++ coding guidelines
- ISO24772 Guidance to avoiding code vulnerabilities
- Japan (code) ESCR guidelines
- MISRA code guidelines
- MISRA guidelines for Automotive Safety Arguments

Current functional safety standards:

- ISO 26262:2018
- DO-178C Avionic Functional Safety Standard
- IEC 61508 Functional Safety Standard
- ISO 29119 Software Testing Standard
- ISO DTR 4804 Road vehicles – Safety and cyber-security for automated driving systems – Design, verification and validation methods
- PAS 1881 Assuring safety for automated vehicle trials and testing
- SAEJ 3061 Cyber-security

Emerging safety standards:

- UL4600 draft standard
- PAS/CD 21448 Safety of the Intended Functionality – SOTIF
- ISO 21434 Automotive Cyber-security standard
- IEEE P2864 – A Formal Model for Safety Considerations in Automated Vehicle Decision Making
- IEEE P1228 – Standard for automated driving software safety
- IEEE P2851 – Exchange/Interoperability Format for Safety Analysis and Safety Verification of IP, SoC and Mixed Signal ICs

## 2.6 Classical computer vision development

Computer vision, or machine vision, has been researched since the 1960s. For decades, computer scientists have been trying to make computers understand images. The research primarily followed a trial and error approach. Algorithm developers manually identified what kinds of features, such as colors, lines and gradients are salient for the specific problem. Groups of detected features were then used as input for classical machine learning algorithms such as Support Vector Machines, Adaboost, and random forests to try to recognize the objects in the image. Viola and Jones in 2001 published an algorithm according to these principles that was widely used for many years to perform face detection for instance [12]. And Dalal and Triggs published a pedestrian detection algorithm in 2005 that lead the field for many years [13]. This process of defining the kinds of features that the algorithm should look for in the image has largely been superseded by AI-based methods, which automates this effort. We will describe some of these in more detail in the next section.

Some fields of classical computer vision are still widely in use though and have not been superseded by AI. For instance, in applications that estimate both the camera's own exact location, as well as extract depth information or 3D information such as point clouds from sequences of images. These Simultaneous Localization and Mapping (SLAM) algorithms track features from frame to frame, and based on their displacement, structure can be extracted using triangulation [14]. This is one example of a problem that can be efficiently solved using predictable, well understood mathematical techniques that consume fewer compute resources than the neural networks used by modern techniques.

### 2.6.1 *Proprietary porting*

Since most of these computer vision algorithms are primarily designed by hand, and not trained as with machine learning-based approaches, they are expressed in a programming language such as C, and not at higher abstraction levels. These programs then need to be mapped and heavily optimized by hand for complex heterogeneous computer architectures. The algorithms often need to be partitioned across heterogenous processors. Besides standard CPU cores, target architectures often include multicore vision processors that employ VLIW (Very Long Instruction Word) and SIMD (Single Instruction Multiple Data) processing, and complex memory hierarchies. The optimization task requires a deep understanding of the underlying computer architectures. Having to meet constraints such as running the algorithms at real-time frame rates of, for instance, 30 frames per second, further complicates this task. This is a very labor-intensive, error-prone, and the resulting implementations are specific to the selected processor microarchitecture, with each semiconductor vendor having their own proprietary and unique microarchitectures. Vendor-specific programming toolchains are also often used, further complicating this task. As a result, the computer vision software that took a lot of work to develop, cannot easily be ported and reused between one architecture and another.

## 2.7 AI-based software development

In 2012, AI research had a major breakthrough when Alex Krizhevsky and his team combined three new technologies together [15]. First, the ImageNet database of images had been released, providing a collection of millions of images that were each described by hand. For each image, a corresponding file labels the objects that are present in the image. Second, Krizhevsky used powerful programmable GPUs, which had recently come on the market, to run his algorithms. This provided orders of magnitude more computational power, opening up the road to using more compute-heavy algorithms.

Third, they developed a new convolutional neural network with many layers, which they called AlexNet. Instead of computer scientists trying to figure out by hand which features to look for in an image, as in classical computer vision, AlexNet figured out what features to look for by itself. AlexNet entered the leading ImageNet Large Scale Visual Recognition Challenge (ILSVRC) image recognition competition and beat the whole field by a whopping 41% margin. Since 2012, every algorithm that has won the ILSVRC competition has been based on similar ”deep learning” principles.

This approach follows two steps: the design and training of the neural network, followed by inferencing, where the network performs the actual task of recognizing the objects in view. We will provide more detail on each of these steps below.

### 2.7.1 *Network design and training*

This step starts with collecting lots of sensor data in the vehicle. To have thousands of kilometers of driving data available is quite common. The dataset can be augmented with synthetic data that comes from simulation, or by manipulating the raw sensor data itself. The following step is to label the data. During this labor-intensive step, the collected data is categorized and annotated by hand, often helped by tools that partially automate the task. Since a single camera takes about 100,000 pictures per hour, labeling several hours of driving data takes large teams of labeling staff. Neural network engineers then design the network architecture: the neural network layers and their configuration, how they are connected, the number of layers, what activation functions to use, the compute accuracy, etc. Automotive applications then combine several of these networks to implement the required complete top-level functionality. During training, the dataset is run through the neural net several times, each time adjusting the coefficients until the neural net's detection performance does not improve any more. This training is performed offline in a data center, using high-performance servers, since the compute requirements are very high. It can easily take hours or even days to train a single network. The output of this process is a trained neural network, which includes a neural network configuration together with the network coefficients that define the operation of the network. This trained network can readily be deployed in the vehicle. Since the AI algorithms are still heavily researched and improved upon, and additional sensor data is being collected all the time, this process of adjusting the network architecture and retraining with new data should be continuous and not stop. The task of data curation becomes important also. There is considerably more data collected than what the networks can be trained with within a reasonable time, so the right subset of data must be selected. In addition, the trained networks need to be validated with data that the trained neural network has not seen yet, to ensure the network can sufficiently generalize to new live sensor data of the vehicle on the road. The datasets are constantly revised with new data from vehicles on the road, better understanding of where the trained algorithms fail, and according to new algorithms with different requirements. Excellent starting point tutorials covering the practical flow of training and inference in TensorFlow$^{TM}$ are provided in [16], e.g., jupyter notebooks, executable in Google co-lab$^{TM}$ on overfitting available in [17].

A relatively recent topic of research and development is self-supervised algorithms. This is a form of unsupervised learning where the data itself provides the supervision. In general, this approach withholds some part of the data, and tasks the network with predicting it. For instance, if a vehicle is detected in a frame, then the vehicle should probably also be present in the previously captured frame. This has the potential to greatly reduce or even remove the reliance on labeled data, resulting in algorithms that have higher accuracy as well as reducing the labor-intensive task of labeling data. Moreover, generative adversarial networks (GANs) **??** are emerging where two concurring networks operate in a competitive manner: The generative network generates candidates whereas the discriminative network evaluates them.

### 2.7.2 *Inferencing*

Once the algorithms are designed and trained offline, they are ready for deployment in the vehicle. This requires the algorithms to run fast, in real-time, and at much lower cost than for training. Since it would take too much bandwidth and latency to transfer the images from the vehicle to the data center, every vehicle has to run the algorithms at the edge, on processors inside the vehicles themselves. The need for low cost, low power consumption, despite the algorithms requiring a lot of compute power to run in real-time, has caused the design of a new class of processors that are specifically designed to run neural network inference.

One big benefit to the developer is that the neural network algorithms are expressed at a high-level of abstraction. The network configuration basically defines a set of filters for each of the neural network layers and the dataflow between them as a graph. The compute primitives to run the network consists of just a few simple operations, primarily multiply-accumulate (MAC), which are endlessly repeated and can easily be parallelized.

Mapping these AI algorithms onto the specialized target hardware is usually a highly automated approach. Most AI processor vendors provide software tools that read in the neural network description and parallelize and optimize the network for running on the accelerator. The net effect is that end users do not have to write a single line of code to run their trained networks on the target device. As an added benefit, this also means that it is much easier to add more complex neural networks that run on one architecture to the next.

Fig. 5 illustrates the first stage of the AI development flow (training) which obviously requires the availability and maturity of the sensor system (sensor/lens/ISP/software/...).

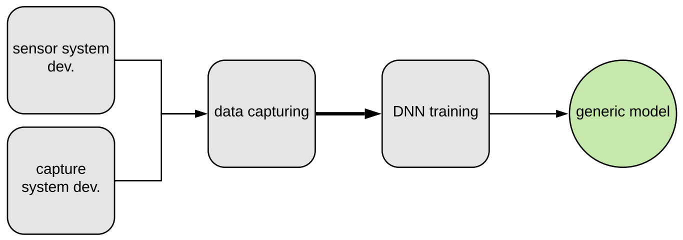


Fig. 5. Generic flow of AI model development that is to be used for inference.

### 2.7.3 *Quantized computing (efficiency mapping)*

Fig. 6 depicts the transfer to the proprietary target using non-standard/proprietary tools. First, the toolchain coverage is limited and often includes manual and error prone steps. Safety context is lost. In addition, the porting to the target implicitly requires certain approximations (e.g. transformation to fixed point, reduction of bit depth) with often unforeseeable impact due to the non-continuous nature of modern (discriminative) deep neural networks (DNNs). There is no decoupling between the development of tools and project specific tasks. Instead, the challenges must be solved along the way.

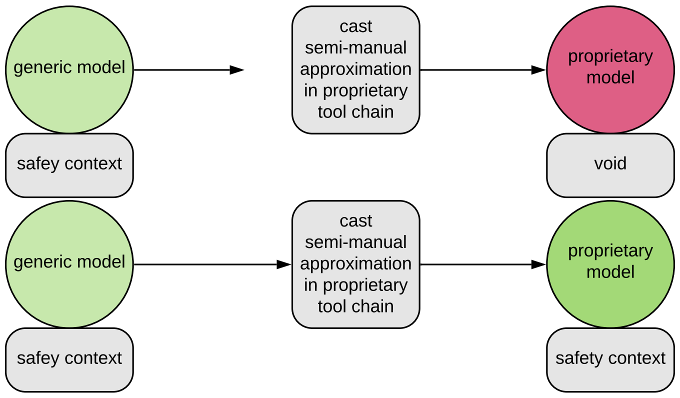

generic model
cast semi-manual approximation in proprietary tool chain
proprietary model
safey context
void
generic model
cast semi-manual approximation in proprietary tool chain
proprietary model
safey context
safety context

Fig. 6. **Top:** transfer gap. **Bottom:** Safety context is preserverd.

According to [18] approximative computing is a major research trend to target this in a scientific and structured way instead.

## 3 Challenges in current Software development for automotive applications

As software complexity increases exponentially the aim is to ideally reduce the development time. With the expected increase in OTA software updates, software development for applications has recently become one of the most challenging and expensive portions of the car-line production. Extensive safety, quality, and reliability requirements remain following the traditional automotive constraints still being necessary. This impacts the pace of innovation and cost, putting pressure on development teams to deliver. Also, the increasing amount of software comes along with an increasing complexity, architecture shaped by particular automotive requirements, such as hard-real time, high reliability and safety, limited resources, cost pressure, short development cycles and heterogeneity of domain knowledge [19].

### 3.1 Development process: V-model

With regard to AutomotiveSPICE (or A-SPICE) V-Model in Fig. 7, the standard process in automotive development and production, software application development is just a single box of SWE.3 “Software Detailed Design and Unit Construction” but porting is done within SWE.4 “Software Unit Verification” with regard to the entire production process. There are many other components to fulfill, including system, hardware requirement, quality assurance, test and validation. The full cycle might take up to three years of development. In reality, the advancing of software solutions is updated quickly with a significant increase in its complexity. This means that there is no available and fixed solution in the rapid growth of software technology. This requires a new strategy of development to catch up with rapid and complex changes. Indeed, agile development has become a hot topic in automotive for years to adapt the practice. Still, with the current development process (V-Model), agile development can only cope with small changes or deviations from the solution path, but this is not efficient for a solution switch nor an upgrade. For a complex problem, if the solution is not fixed, then there must be more iterations inside the development cycle where Software-in-Loop (SIL) and Hardware-in-Loop (HIL) have to be performed continuously together with software modifications and updates. In reality, typical SIL and HIL activities might take years to be done with a large dataset, meanwhile an upgrade of a state-of-the art solution might already be required. This enforces a mutual development or a very close collaboration between the advanced and production development. An example of V-Model+ (or modified V-Model) to be more flexible with the agile development idea is illustrated in Fig. 8. In this new process, two loops are designed for possible updates or upgrades of software:

1) the first loop is to continuously update the solution to satisfy given KPIs at SIL;
2) verification of the updated solution in the target hardware.

Advanced development is expected to join the first loop and the second loop partially at the small scale of dataset. The production development will be responsible for the modification and verification on a much larger dataset than usual.

### 3.2 Human resource demand and Operational Challenges

Achieving very high-quality assurance and safety targets drives the automotive world to become quite a closed community and difficult to join for newcomers. Additionally, many aspects of development can only be covered by highly experienced members (10 to 20 years of hand-on experience), so finding trained personnel is always a challenge. This traditional issue comes from the variety of highly customized and optimized automotive systems which need very much hands-on experience at production level. A clear path to solve the issue is to approach more advanced standards and advanced technologies which are generic enough and familiar to the latest generation of programmers.

#### *3.2.1 A new kind of programmer is needed*

A developer who is versed in standard modern C++ can use SYCL very quickly. The challenge for a developer is to shift their thinking to conceptualize programming in a highly parallel programming paradigm. It requires practice and familiarization with new programming and design approaches in order to create efficient programs. There are also parallel libraries that are emerging to reduce the coding effort required by the programmer, like Parallel STL (C++ Standard Template Library). With parallel programming comes new programming design patterns, which have existed in the HPC domain for years. Patterns, like the reduction pattern, have been crafted over the years to maximize performance and efficiency on parallel computer systems such as supercomputers. These patterns need to be coded efficiently, to reduce power consumption and improve execution time in any computing application. The developer is free to add additional libraries to their application, even non-C++ libraries like Python with few restrictions. Python is used here to express tensor models used by Google$^{TM}$'s TensorFlow$^{TM}$ library (written in C++).

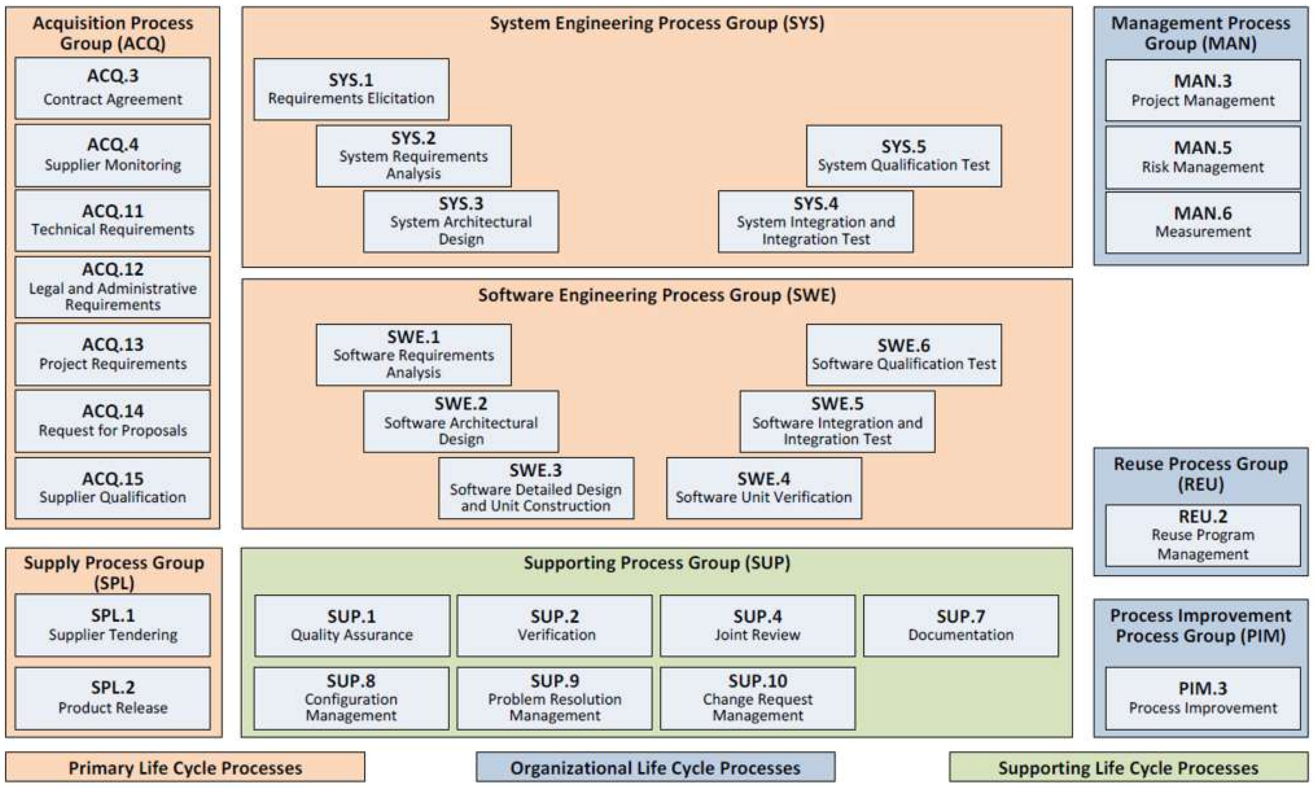

Acquisition Process Group (ACQ)
ACQ.3 Contract Agreement
ACQ.4 Supplier Monitoring
ACQ.11 Technical Requirements
ACQ.12 Legal and Administrative Requirements
ACQ.13 Project Requirements
ACQ.14 Request for Proposals
ACQ.15 Supplier Qualification
System Engineering Process Group (SYS)
SYS.1 Requirements Elicitation
SYS.2 System Requirements Analysis
SYS.3 System Architectural Design
SYS.4 System Integration and Integration Test
SYS.5 System Qualification Test
Management Process Group (MAN)
MAN.3 Project Management
MAN.5 Risk Management
MAN.6 Measurement
Software Engineering Process Group (SWE)
SWE.1 Software Requirements Analysis
SWE.2 Software Architectural Design
SWE.3 Software Detailed Design and Unit Construction
SWE.4 Software Unit Verification
SWE.5 Software Integration and Integration Test
SWE.6 Software Qualification Test
Reuse Process Group (REU)
REU.2 Reuse Program Management
Supply Process Group (SPL)
SPL.1 Supplier Tendering
SPL.2 Product Release
Supporting Process Group (SUP)
SUP.1 Quality Assurance
SUP.2 Verification
SUP.4 Joint Review
SUP.7 Documentation
SUP.8 Configuration Management
SUP.9 Problem Resolution Management
SUP.10 Change Request Management
Process Improvement Process Group (PIM)
PIM.3 Process Improvement
Primary Life Cycle Processes
Organizational Life Cycle Processes
Supporting Life Cycle Processes

Fig. 7. A-SPICE V-Model

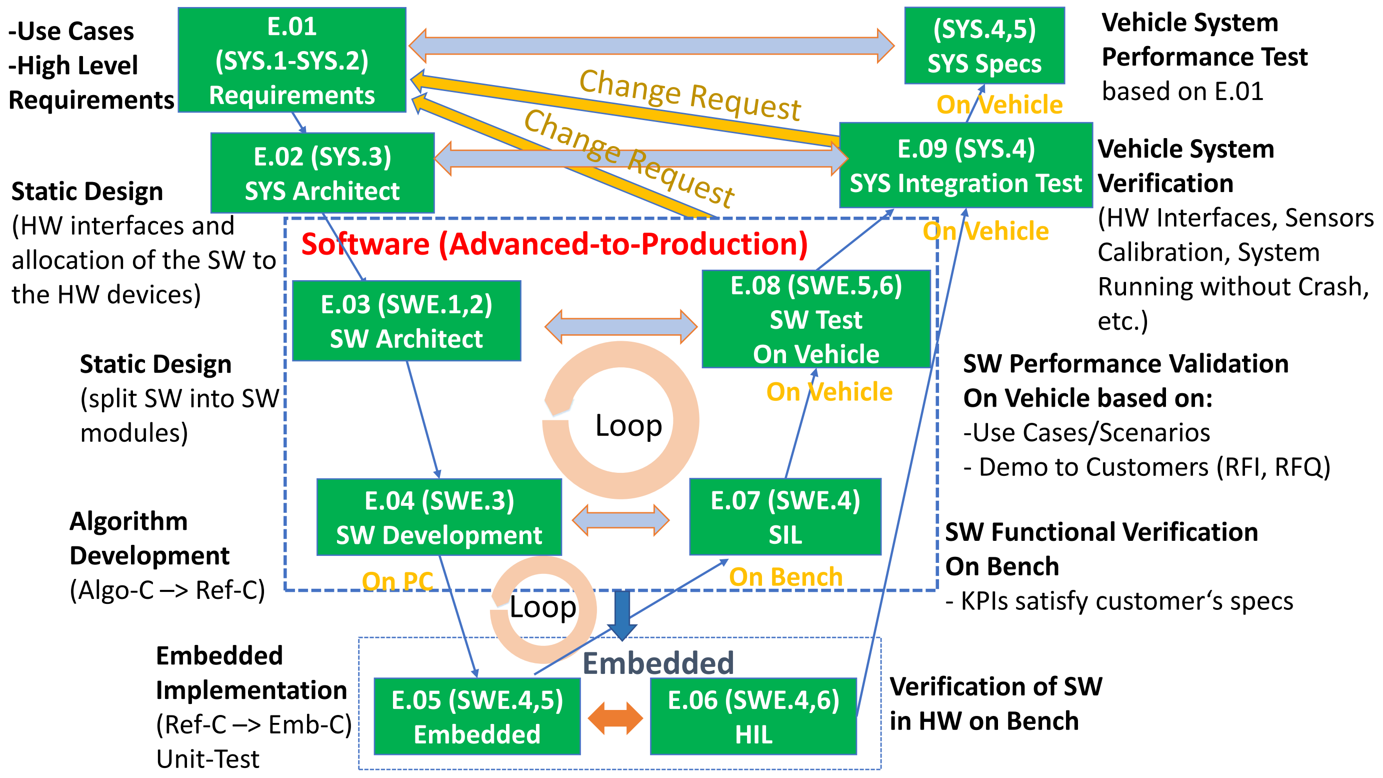

-Use Cases
-High Level Requirements
E.01 (SYS.1-SYS.2) Requirements
(SYS.4,5) SYS Specs
On Vehicle
Vehicle System Performance Test based on E.01
Change Request
Change Request
E.02 (SYS.3) SYS Architect
E.09 (SYS.4) SYS Integration Test
On Vehicle
Vehicle System Verification (HW Interfaces, Sensors Calibration, System Running without Crash, etc.)
Static Design (HW interfaces and allocation of the SW to the HW devices)
Software (Advanced-to-Production)
E.03 (SWE.1,2) SW Architect
E.08 (SWE.5,6) SW Test On Vehicle
On Vehicle
Loop
SW Performance Validation On Vehicle based on:
-Use Cases/Scenarios
- Demo to Customers (RFI, RFQ)
Static Design (split SW into SW modules)
E.04 (SWE.3) SW Development
E.07 (SWE.4) SIL
On PC
On Bench
SW Functional Verification On Bench
- KPIs satisfy customer‘s specs
Algorithm Development (Algo-C –> Ref-C)
Loop
Embedded
Embedded Implementation (Ref-C –> Emb-C) Unit-Test
E.05 (SWE.4,5) Embedded
E.06 (SWE.4,6) HIL
Verification of SW in HW on Bench

Fig. 8. Revolutionized V-Model for Agile Development

A developer only versed in C++ prior to C++ '11 will have to learn some modern C++ features required to code with SYCL. An example of this type of developer is one who has been writing C++ following the automotive industry's de facto coding standard, MISRA C++. They would also need to learn to use C++ templates and some more general good practices like RAII (pattern Resource Acquisition Is Initialization) or familiarize themselves with the supporting STL resource management functions.

A developer only versed in 'C' programming operating close to the hardware, using global variables and poking registers, will have a steeper learning curve. They would need to familiarize themselves with object-oriented programming (OOP) and practice writing C++ using all its features like class inheritance, and templates while taking care not to continue writing 'C' style code. With that also comes learning all the additional set of C++ programming complexities on top of the existing 'C' ones.

A comparison of different levels of standards and their decoupling/applicability to hardware, software and libraries is depicted in Fig. 9.

According to the Khronos Group [20]: "SYCL is a standard C++ based heterogeneous parallel programming framework for accelerating High Performance Computing (HPC), machine learning, embedded computing, and compute-intensive desktop applications on a wide range of processor architectures, including CPUs, GPUs, FPGAs, and AI processors."

### 3.3 Chip dependency

Due to cost and performance restrictions, automotive hardware is highly specialized and optimized for some specific use-cases. It is often not flexible, and sometime not even programmable where developers can only call existing functions with some configurable parameters. This severely restricts the scalability, as well as the applicability of new innovative solutions. Indeed, most state-of-the-art technologies could not be ported to the deployed platform due to many hardware limitations. For example:

- some (like deep learning) require too much throughput which is out of scope of the current hardware.
- some (like SLAM) do not require a powerful processing unit, but need a fast memory copy between CPU and GPU, which is not supported by most of automotive hardware.
- some (pixel processing, e.g., like Optical Flow) are performed by hardware acceleration with compromised precision, which do not allow developers to implement their desired/updated/innovative algorithm. Also, low-level pixel processing requires the support of a hardware acceleration unit, which is heterogeneously designed by different companies. Developers might even need to learn a new type of low-level programing language for specific hardware, such as EVE, FPGA, etc.
- some (like sensor fusion, or optimization solvers) need a high precision-resolution, which is strongly constrained by hardware.

With all the examples of hardware dependency, it is obvious that switching to new hardware almost always needs software development to start again from scratch. Reuse or CD is not possible with such a hardware limited dependency.

### 3.4 Handling software bugs

The recent increase in call-backs from OEMs for bug-fix issues might be explained by the reduction of development time before production, while the complexity of the systems increases significantly. In fact, heterogeneously integrated design solutions might account for further issues, e.g., some parts inherited from SoC libraries, and some coming from Tier 1 development. Also, many available solutions are designed to work under many narrow assumptions which are easily failed in real-life scenarios, for example a perfectly flat road, structured environment, etc. Design issues have no easy fix, and usually lead to a failure of the product. Alternatively, the majority of small bugs can be easier to fix by applying new software update. The main cause for this issue should account from insufficient quality assurance, test and validation. Due to the cost and time pressures, not many Tier 1/Tier 2 or even OEMs are able to strictly apply the A-SPICE 3 standard or don't fully perform functional safety. A modern approach for the later issue type is to have OTA software updates. This is somewhat controversial and is often referenced as potentially impacting the safety and security of the vehicle when not fully verified.

## 4 WHERE IS THE NEW SOFTWARE COMING FROM?

An important trend that has been observed recently is the shift towards powerful, centralized controllers. Whereas vehicles in the past used hundreds of small ECUs, each typically devoted to a single task, modern vehicles use a smaller number of more powerful centralized domain controllers. For example, Tesla® uses a single domain controller and the Volkswagen® ID.3 uses two controllers.

As these use centralized domain controllers, they are now responsible for the execution of mainly simultaneous tasks, and also need to fulfill real-time constraints for those tasks that have such requirements. These controllers need to provide vast amounts of computational power, which is typically not only achieved by scaling up the number of processor cores, but also through the deployment of *heterogeneous* systems, combining CPUs with a variety specialized processors and accelerators.

Examples of such accelerators include, next to GPUs, FPGAs or custom processors for image processing or AI tasks. Programming such heterogeneous systems with a variety of different functional units poses new challenges to the automotive industry, which has so far mainly relied on inherently sequential programming models. To make full use of the computing capability provided by platforms, the automotive industry needs to adopt programming models that allow it to fully leverage the parallelism and variety of specialized accelerators on such platforms. However, there is no need to re-invent the wheel, parallel and heterogeneous programming models have been researched and established in other sectors such as smartphones, PCs and *High-Performance Computing* (HPC), with many of the models serving as a good starting point for automotive software development. Yet, the selection of one or more multiple programming models, coupled with proprietary vs. open standards, is a crucial decision.

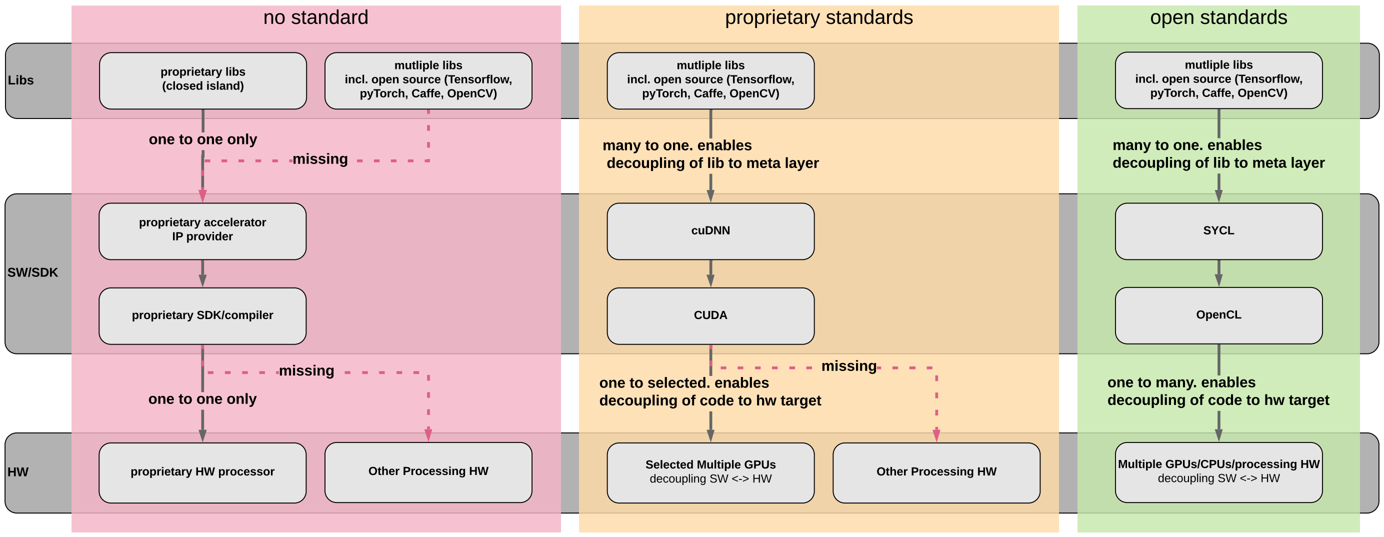

no standard
proprietary standards
open standards
Libs
SW/SDK
HW
proprietary libs (closed island)
mutliple libs incl. open source (Tensorflow, pyTorch, Caffe, OpenCV)
one to one only
missing
proprietary accelerator IP provider
proprietary SDK/compiler
proprietary HW processor
Other Processing HW
many to one. enables decoupling of lib to meta layer
cuDNN
CUDA
one to selected. enables decoupling of code to hw target
Selected Multiple GPUs decoupling SW <-> HW
SYCL
OpenCL
one to many. enables decoupling of code to hw target
Multiple GPUs/CPUs/processing HW decoupling SW <-> HW

Fig. 9. Comparison of no standards vs. proprietary vs. open standards w.r.t. independence towards hardware, software and high-level libraries

## 4.1 Proprietary

One of the most popular proprietary programming models is the CUDA programming language, developed and maintained by Nvidia. CUDA allows the efficient programming of GPUs in heterogeneous systems with a programming language based on C/C++. The survey in [21] found the CUDA ecosystem, including compilers, profilers and specialized libraries for tasks such as AI training and inference, to be very rich. In comparison to the open standard OpenCL and its diverse hardware support, the focus of CUDA is solely on officially supported Nvidia devices using Nvidia GPUs. This does however allow for a more consistent development environment with good performance. A CUDA program still requires restructuring to enable parallelism and to offload computations quickly.

However, CUDA, just like any other proprietary standard, has one major drawback, the danger of a potential vendor lock-in. Production-ready compilers are only available from Nvidia and only for Nvidia devices (currently only GPUs). This puts one of the most important assets of an automotive company, namely its production software code, at a high risk, because a switch to a vendor other than Nvidia could potentially render the entire existing CUDA code base almost useless. Besides that, proprietary programming models are typically only available on a very limited set of devices. In the case of CUDA, the programming model and API can only be used to target Nvidia GPUs, but not to target other important components of heterogeneous systems, such as multi-core CPUs, FPGAs or specialized, custom accelerators for AI.

## 4.2 Open standards

A key advantage of an open standard lies in its diversity: It takes many experts from many different companies and fields (each with different experiences) to design the specification of an API. Thus, with so many interested parties working together for mutual benefit, improvements can be made quickly to address concerns and turn around a revised specification. Open standard implementations are available from different vendors. The software API provides their products with a common programming interface for a large variety of devices and architectures. One example of such an established standard available on a broad range of devices is OpenCL. OpenCL is not only used by most GPU vendors, but can also be used to program multi-core CPUs, FPGAs (both Intel® and Xilinx®) and more specialized processors (e.g. Renesas® R-Car).

Another example of a well-established, successful standard for parallel programming is OpenMP. Although OpenMP in the beginning was mainly designed as a shared memory programming model for homogenous multi-core CPUs, it has been extended to also target heterogenous accelerators in recent versions of the standard. Nowadays, OpenMP device offloading support is available for GPUs from many different vendors, including AMD®, Nvidia® and Intel®, and for other specialized accelerators such as vector engines.

In 2014 SYCL became a new addition to the body of open, parallel programming models using modern standard C++ and the single source programming model. Designed as a spiritual successor to OpenCL, SYCL avoids many of the pitfalls of OpenCL identified in earlier studies (e.g. [21], [22]), such as the need to distribute host and device code across different compilation units or the extremely verbose host API. SYCL is also designed to interoperate extremely well with C++ as the underlying programming language and its mechanisms. This allows developers to reuse significant parts of existing, sequential implementations, which has proven to be extremely useful to avoid introducing errors into the code during parallelization in the context of OpenMP [21].

Although SYCL is a relatively new standard, it has received increasing attention and a variety of vendors have announced or already released SYCL support, including CPU- & GPU-vendors (AMD® ROCm, Intel®), FPGA-vendors (Xilinx®, Intel®) and others, e.g., Renesas®/Codeplay for R-Car. This broad adoption and the availability for many different platforms and architectures make SYCL a very interesting candidate for future development of automotive software. Section 6 will discuss SYCL in more detail.

Common to all open standards mentioned in this section is that they are maintained by multi-partisan institutions, e.g., the Architecture Review Board (ARB) or the Khronos working groups for SYCL and OpenCL. These institutions, which develop and maintain the open standards, offer a well-suited platform for stakeholders to engage in the development of these standards and provide an opportunity for the automotive industry to raise awareness for their particular requirements.

### 4.3 Open-source safety concern

The computing stack evolves, some software becomes ”legacy,” algorithms get improved, and new algorithms are found to replace the ones that are no longer suitable. In the dynamic world where there are few considerations for safety requirements, the technology and its software will evolve very fast, and this is a good thing as it drives innovation forward. Faster innovation and iterative development are methods that the automotive industry is seeing as a way to develop the next generation of vehicle features that consumers want and regulations stipulate. A lot of the open-source projects being used by researchers are not developed following functional safety standards, but they are likely to evolve to be usable in a safety critical environment too. Once this technology is brought into the automotive domain, the speed of evolution, that was delivering the rapid technology research, stalls as it has to follow safety processes and meet specific operational safety requirements. The developers who work in open-source communities do not necessarily have a vested interest in safety applications, and while they help evolve the software, they may have no consideration for supporting software for safety purposes. The burden of work now falls on the safety application developers to evolve the tools and software, often with a much smaller number of people.

## 5 Software complexity is growing

Automotive vehicles are known to be the most complex mass product of mankind, with the innovations rising exponentially. Additional degrees of complexity are introduced by the ever-growing number of car lines and car line variants. Where, decades ago, an OEM focused on three car lines (e.g. Mercedes® C-Class / BMW® 3 series, E-Class / 5 series, S-Class / 7 series) now a huge and ever-increasing variety is available. The same applies to the option variants within each car line. The increasing connectivity and interaction with other systems of the same vehicle instance further adds degrees of complexity. In addition, the design cycles of subsystems/components differ, so the set of combinations is further fueling the complexity degrees of freedom and this leads to ”uncontrolled growth” and fragmentation. According to Conway’s rule, the resulting ADAS are limited by the given organization/communication structure of the company [23] : ”Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication structure” [24].

### 5.1 How is this managed

The automotive industry is applying several quality measures to control the software complexity and to manage software development.

### 5.2 CI/CD challenges

#### *5.2.1 What are they?*

- Breaking software APIs APIs (Not following an agreed open stand)
- Breaking Application Binary Interfaces for libraries (Not adhering to C++ guidelines to prevent issues)
- The archiving, management and transportation of large volumes of applications’ data
- The manual elements of CI / CD for automotive
- Multiple vendors in the development of the application
- Open-source and development direction to support automotive’s requirements

CI means a high frequency of updates to all parts of the ADAS application compute stack.

One of the attractions of open standards and the API specification is that they ensure an application can use any implementation which has shown to be conformant. This means for the developer that their applications do not need to be rewritten for different targets. However, new versions of libraries require testing within the stack to provide assurances that applications still operate as expected. A supporting development plan would likely mandate the use of a comprehensive testing suite.

For some developers, the work involved in keeping up-to-date with the latest changes elsewhere in the stack could be too much, and so they may opt to hold back on upgrading. The disadvantage here is, as the developer holds back, the greater the changes they need to make when they adopt a later version. Technology like OTA updates in vehicles is enabling the frequency of updates to increase. This is very attractive, as it allows security updates or bug fixes to get to the vehicles quickly without the need for recalls, added costs and delays, but it does lead to concerns of how to validate such complex software in short periods of time.

While the open standard ensures conformance between major versions it does not mean implementers’ minor versions are always compatible. There are two ways a 3rd party software library can break an application should it be upgraded:

1) Application facing API (application programming interface) changes by the library
2) Hidden ABI (application binary interface) changes within the library

In a large C++ codebase with many internal interfaces between components, whether developed in house or by the open source community, the opportunities to break binary compatibility between versions are high. The source of these incompatibility changes come from the use of the C++ language and how the C++ compiler compiles the code. Many of the C++ breaking code changes can be identified quite easily and, with the use of good coding development policies in place, developers can avoid ABI incompatibility changes.

The application developer invests time and gains confidence in the libraries they choose to use. When API or ABI changes break their application, they must divert resources to investigate issues and this, if it occurs too frequently, dents confidence. The frequency of library changes and fixes means many more new releases of the applications into the field. An implementation supporting semantic versioning would allow the practitioners to trace changes to vehicles.

## 6 SYCL

SYCL is a promising candidate to address the challenges of programming parallel and heterogeneous systems in the future. It is intended to support diverse applications from HPC and embedded applications, to powerful frameworks for machine learning [25], [26], [27]. The open standard, backed by many companies across a variety of platforms, such as CPUs, GPUs and FPGAs, is designed to bring parallel and heterogeneous programming capabilities to C++ and eventually converge with ISO.

As outlined in [28], SYCL does not require any additional language constructs such as pragmas and relies on pure C++. This also allows the re-use of an existing, serial code base as a starting point for a parallel implementation and requires no extensive restructuring of the code, two properties of programming models that proved highly beneficial in the study in [21]. The SYCL runtime will organize the different kernels into a task-graph, based on the data-access specification, and will handle scheduling, synchronization and data management automatically. The kernels themselves use a data-parallel execution model, similar to OpenCL or CUDA kernels.

### 6.1 SYCL training and usage

As highlighted, SYCL is evolving as an industry standard for heterogenous programming. Substantial training material is available. As a main source of information, the SYCL ecosystem website is available – *www.sycl.tech*. Here, the latest updates, research activities and code can be found. For programmers, additional information and training can be found in Codeplay's SYCL guide [29].

The generic flow with standardized interfaces is illustrated in Fig. 10.

Since safety is a crucial element for automotive, a series of articles providing guidance for Functional Safety Managers and developers, ”SYCL for Safety Practitioners”, is being published. An introduction to the guidance material is given in [31].

### 6.2 SYCL status quo

Although the SYCL open standard specification made its first appearance in 2014, it is still steadily evolving to meet the needs of its Khronos work group members. SYCL is gaining the interest of many industries such as automotive, HPC and others. Intel recently embraced SYCL as the core of its oneAPI initiative (Intel's SYCL implementation and supporting tools), feeding additional requirements into the standard. One of the main concerns of the industry,

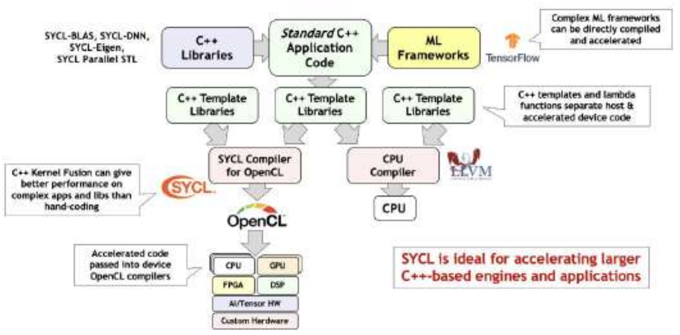

SYCL-BLAS, SYCL-DNN, SYCL-Eigen, SYCL Parallel STL
C++ Libraries
Standard C++ Application Code
ML Frameworks
TensorFlow
Complex ML frameworks can be directly compiled and accelerated
C++ Template Libraries
C++ Template Libraries
C++ Template Libraries
C++ templates and lambda functions separate host & accelerated device code
SYCL Compiler for OpenCL
CPU Compiler
LLVM
C++ Kernel Fusion can give better performance on complex apps and libs than hand-coding
SYCL
OpenCL
CPU
Accelerated code passed into device OpenCL compilers
CPU
GPU
FPGA
DSP
AI/Tensor HW
Custom Hardware
SYCL is ideal for accelerating larger C++-based engines and applications

Fig. 10. Layers of implementation from [30] enabling the decoupling of proprietary HW to enable true continuous development over multiple generations and platforms.

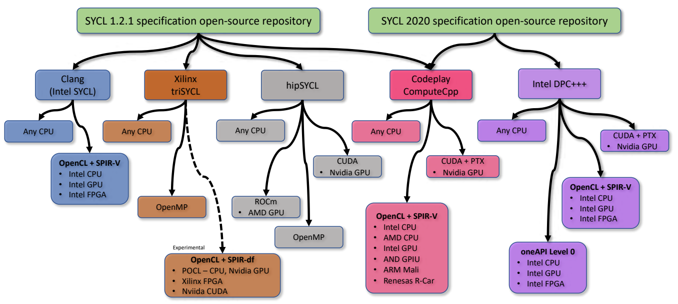

SYCL 1.2.1 specification open-source repository
SYCL 2020 specification open-source repository
Clang (Intel SYCL)
Xilinx triSYCL
hipSYCL
Codeplay ComputeCpp
Intel DPC+++
Any CPU
OpenCL + SPIR-V • Intel CPU • Intel GPU • Intel FPGA
Any CPU
OpenMP
Experimental
OpenCL + SPIR-df • POCL – CPU, Nvidia GPU • Xilinx FPGA • Nviida CUDA
Any CPU
CUDA • Nvidia GPU
ROCm • AMD GPU
OpenMP
Any CPU
CUDA + PTX • Nvidia GPU
OpenCL + SPIR-V • Intel CPU • AMD CPU • Intel GPU • AND GPIU • ARM Mali • Renesas R-Car
Any CPU
CUDA + PTX • Nvidia GPU
OpenCL + SPIR-V • Intel CPU • Intel GPU • Intel FPGA
oneAPI Level 0 • Intel CPU • Intel GPU • Intel FPGA

Fig. 11. There are different implementations available supporting different platforms already. Further SoC and IP vendors are following.

especially in the automotive context, is power and silicon efficiency. As discussed earlier, this has so far been one of the key decision-making criteria for new chip and underlying silicon accelerator IP. There are different implementations available, as can be seen in Fig. 11 from [30]. They all follow the open and public standard with different layers running on different hardware underneath, as shown in Fig. 12. Moreover, SYCL kernels can run on CPUs as well as multiple targeted, discrete devices, like GPUs, DSPs or FPGAs, at once which further increases its performance and portability capabilities. Another factor the SYCL specification has addressed in its design and proved successful in is the ability to manage data movement between the host and target devices in a manner that is unintrusive for the developer. SYCL manages the multiple data dependencies between the host and devices and synchronizes data once the developer commits kernels for execution. An interesting point for a SYCL adopter is, while there is a SYCL specification, the adopter or SYCL implementation developer does not have to follow the specification exactly. The implementation can be customized to a customer's requirements which can mean a lighter implementation if necessary. A SYCL implementation must follow the Khronos specification if the product is to be certified and marketed as having a SYCL implementation.

There are several performance figures and benchmark activities available which focus on specific use cases and scenarios. [32] shows very comparable performance results between SYCL and CUDA. In [33] it is shown how highly parametrized SYCL kernels have a competitive advantage over manually tuned libraries such as clBLAST and others.

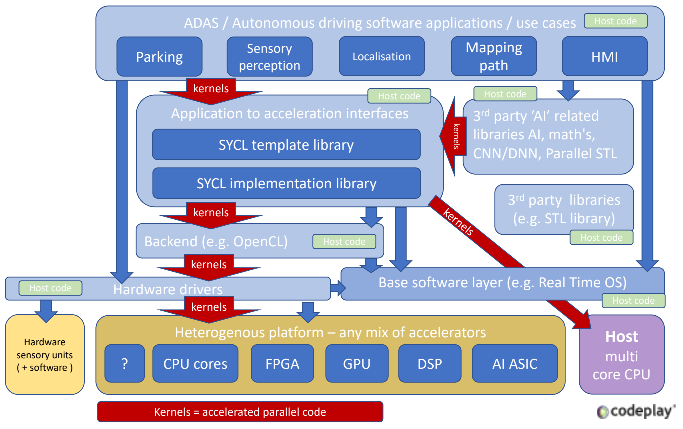

ADAS / Autonomous driving software applications / use cases
Host code
Parking
Sensory perception
Localisation
Mapping path
HMI
kernels
Host code
Application to acceleration interfaces
SYCL template library
SYCL implementation library
Host code
3rd party 'AI' related libraries AI, math's, CNN/DNN, Parallel STL
kernels
3rd party libraries (e.g. STL library)
Host code
kernels
Backend (e.g. OpenCL)
Host code
kernels
kernels
Host code
Hardware drivers
Base software layer (e.g. Real Time OS)
Host code
kernels
Hardware sensory units (+ software)
Heterogenous platform – any mix of accelerators
?
CPU cores
FPGA
GPU
DSP
AI ASIC
Host multi core CPU
Kernels = accelerated parallel code
codeplay

Fig. 12. SYCL (C++ Single-source Heterogeneous Programming for OpenCL) for OpenCL

In [34] a comparison between OpenCL and OpenMP is performed. Here, the conclusion indicates a performance gap for SYCL remains but highlights the potential for further narrowing. An ADAS or AI application developer using SYCL to write kernel programs to accelerate a computationally intensive task through the use of parallelism can have their code execute on any of the target devices shown in Fig. 12 (below each of the different implementations) without the need for rewriting the kernel programs. With SYCL's single source model of application development (kernels and host code side-by-side) portability is key. All SYCL kernel programs will calculate the same results no matter the device it is executed on, although they can vary in execution speed which is expected. In [35] CUDA also outperforms SYCL to date, however optimization for cross platform SYCL applications has just started. The clear conclusion from publicly available data sources indicate that SYCL is very close in performance to proprietary and hardware specific implementations. Demand from the automotive industry, with a clear requirement for a highly optimized implementation, will accelerate progress on this path further.

## 7 Key Technical Requirements for Automotive for Open Standards

Various KPIs are needed within the development cycle to evaluate, quantify and rate the capabilities of automotive software projects. Such multidimensional KPIs require score weightings to summarize to a single score. The weighting is potentially a function of project goals, supply chain roles and user scopes. However, it is strongly felt that the following items are relevant to most of the users in this space with an ever-increasing importance weighting on AI, computer vision and maintainability. A high-level overview of major relevant KPI areas is depicted in Fig. 13.

Not only these technical aspects but also organizational, legal and commercial aspects are relevant to consider and measure. Shorter and shorter development cycles will ultimately lead to the need for continuous development and, crucially, an optimization of development time. While the availability of libraries is crucial, the underlying maturity of them is key. The more people that use such flows, the higher

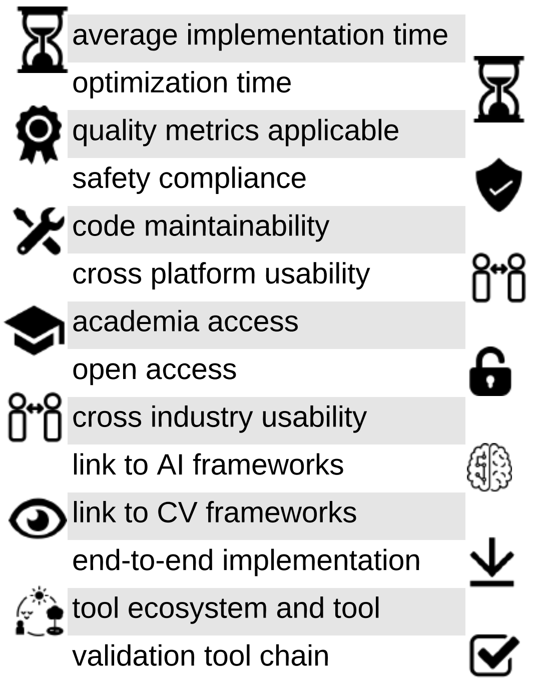


Fig. 13. Relevant high level KPIs for automotive open standards.

the maturity of the development process. The corresponding automotive KPIs are relevant on component/sub-system level but also on system level, as illustrated by gauges in Fig. 14 where each component is measured and strongly contributes to the overall system performance.

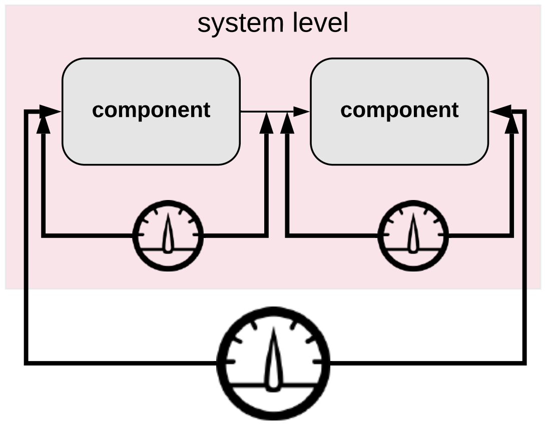


Fig. 14. KPIs need to be applicable for both end-to-end system level quality measurements but also for component/sub-system level domains.

The high-level clustering of relevant KPI categories into mainly technical categories, as opposed to commercially driven KPIs, is also important, as shown in Fig. 15. Of course, all components always have a relevant commercial aspect underneath.

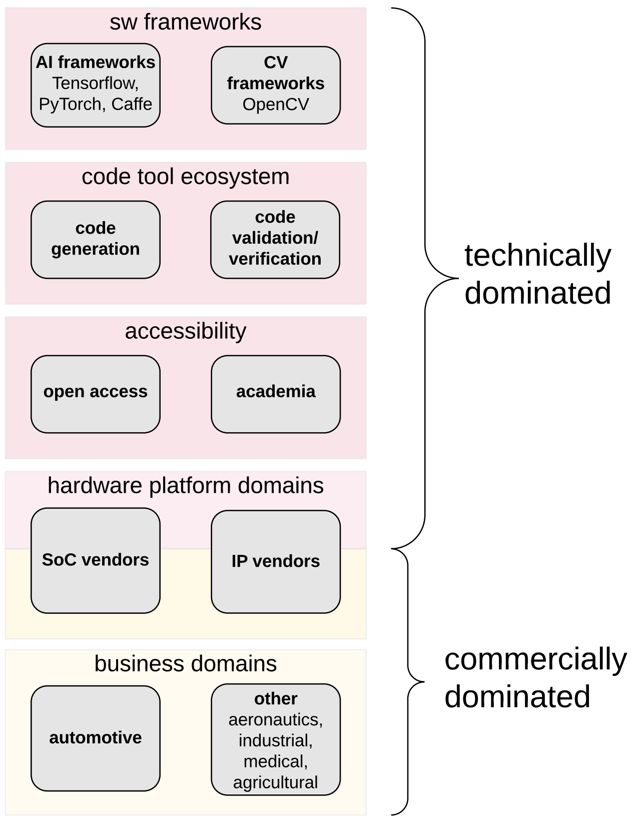


Fig. 15. Clustering in mainly technical vs. mainly commercially dominated KPI categories.

## 8 Gap discussion open standards vs. automotive requirements

### 8.1 What is missing inside SYCL for automotive (if anything) with respect to KPIs

Integration of safety and the Safety-of-the-Intended-Function (SOTIF) approaches have a big impact on software development. First stages are being implemented and rely heavily on standard programming models [36]. Automated code coverage checking, interface testing and other standard procedures that are established in classical "C" are not yet available for fully heterogenous compute environments. The latest SYCL specification focuses on standard use cases. The next step for support and diagnostic routines will have to be agreed by the industry to ensure the exact requirements are captured to enable cross platform availability. Since the exact mechanism depends on the underlying hardware, a generic abstraction formulation must be defined and agreed [30].

With Khronos being an open standards group, any expert is invited to contribute and close the gaps.

## 9 Conclusion and recommendations

In this work, we have outlined the status quo of the vehicle development process in general and the software development process in particular. In the future and with the advent of ADAS functionality, software is becoming a crucial asset of automotive OEMs and Tier 1 suppliers. To manage the complexity of modern automotive software and preserve software IP across vehicle generations, the automotive industry will need to adopt techniques from the Continuous Development (CD) approach.

There is nothing missing today to allow OEMs, Tier 1s, research, advanced developments and software partners to start evaluating and developing with SYCL. The ecosystem is well established, and many organizations are already contributing their results, open source projects and guidance, providing the support for anyone to get up and running.

The question is more about timing than the likelihood of its adoption: when will ADAS software developers transition to a modern C++ programming style. Many companies are already demanding OpenCL or SYCL within their requirements and it is reassuring to see real ambition to compete with the companies investing heavily in autonomous vehicles.

There are still some hurdles to overcome with this approach, we do not pretend it is without issues, but the demand for the latest algorithms and programming methods already embraced by some major companies can only steer development teams towards open-standards-based solutions.

Open standards are crucial to enable the automotive digital transformation and make CD possible. SYCL as an open standard shows absolute potential and is compatible with the CD processes introduced into automotive software development in the future. The SYCL implementations that already exist today are examples of CD in action. As with any open standard, its evolution comes from its contributors. The key stakeholders in the automotive industry can adopt CD, supported by open standards like SYCL, for very little entry cost.

## References

[1] M. Q. Pearl Doughty-White, *Codebases*, 2020 (accessed July 5, 2020). [Online]. Available: https://informationisbeautiful.net

[2] G. FASTR group data from IEEE Spectrum, *The organically secure vehicle of tomorrow*, 2020 (accessed July 5, 2020). [Online]. Available: https://fastr.org

[3] R. McOuat, *Cars are made of code*, 2020 (accessed July 5, 2020). [Online]. Available: https://blog.nxp.com/automotive/cars-are-made-of-code

[4] G. G. Claps, R. B. Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21–31, 2015.

[5] *The case for an end-to-end automotive software platform*, 2020 (accessed July 5, 2020). [Online]. Available: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/the-case-for-an-end-to-end-automotive-software-platform

[6] S. Ravi, *The Audi vision of autonomous driving: A conversation with Thomas Müller*, "https://www.linkedin.com/pulse/audi-vision-autonomous-driving-conversation-thomas-mller-sanjay-ravi/?articleId=6612208053502775296", 2020 (accessed July 5, 2020).

[7] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev, "Compute solution for tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020.

[8] G. Booch, *Object Oriented Design with Applications*. USA: Benjamin-Cummings Publishing Co., Inc., 1990.

[9] M. Fowler and M. Foemmel, *Continuous integration - ThoughtWorks*, 2006. [Online]. Available: http://www.thoughtworks.com/ContinuousIntegration.pdf

[10] T. ROSSI, *Autonomous and ADAS test cars produce over 11 TB of data per day*, 2020 (accessed July 5, 2020). [Online]. Available: https://www.tuxera.com/blog/autonomous-and-adas-test-cars-produce-over-11-tb-of-data-per-day/

[11] S. Heinrich, “Flash memory in the emerging age of autonomy,” *Flash Memory Summit*, 2017.

[12] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, 2001, pp. I–I.

[13] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1, 2005, pp. 886–893.

[14] R. Smith and P. Cheeseman, “On the representation and estimation of spatial uncertainty,” *The international journal of Robotics Research*, vol. 5, no. 4, pp. 56–68, 1986.

[15] A. Krizhevsky, I. Sutskever, and G. Hinton, “Imagenet classification with deep convolutional neural networks,” in *NIPS*, 2012, pp. 1106–1114.

[16] colab.research.google.com, *Tensorflow tutorials*, 2020 (accessed July 5, 2020). [Online]. Available: hhttps://www.tensorflow.org/tutorials?hl=en

[17] ——, *Explore overfitting and underfitting*, 2020 (accessed July 5, 2020). [Online]. Available: https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/keras/overfit_and_underfit.ipynb

[18] W. Liu, F. Lombardi, and M. Shulte, “A retrospective and prospective view of approximate computing [point of view,” *Proceedings of the IEEE*.

[19] V. Schulte-Coerne, A. Thums, and J. Quante, “Challenges in reengineering automotive software,” in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 315–316.

[20] *Khronos overview*, 2020 (accessed July 5, 2020). [Online]. Available: https://sycl.tech/news/20/07/01/sycl2020-released-Khronos.html

[21] L. Sommer, F. Stock, L. Solis-Vasquez, and A. Koch, “Ephos: Evaluation of programming - models for heterogeneous systems,” 2019.

[22] L. Sommer, F. Stock, L. Solis-Vasquez, and A. Koch, “Work-in-progress: Daphne - an automotive benchmark suite for parallel programming models on embedded heterogeneous platforms,” in *2019 International Conference on Embedded Software (EMSOFT)*, 2019, pp. 1–2.

[23] L. J. Colfer and C. Y. Baldwin, “The mirroring hypothesis: theory, evidence, and exceptions,” *Industrial and Corporate Change*, vol. 25, no. 5, pp. 709–738, 09 2016. [Online]. Available: https://doi.org/10.1093/icc/dtw027

[24] M. E. Conway, “How do committees invent?” *Datamation*, April 1968. [Online]. Available: http://www.melconway.com/research/committees.html

[25] R. Burns, J. Lawson, D. McBain, and D. Soutar, “Accelerated neural networks on opencl devices using SYCL-DNN,” *CoRR*, vol. abs/1904.04174, 2019. [Online]. Available: http://arxiv.org/abs/1904.04174

[26] J. Lawson, “Towards automated kernel selection in machine learning systems: A sycl case study,” 2020.

[27] M. Goli, L. Iwanski, and A. Richards, “Accelerated machine learning using tensorflow and sycl on opencl devices,” in *Proceedings of the 5th International Workshop on OpenCL*, ser. IWOCL 2017. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3078155.3078160

[28] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, “Sycl-bench: A versatile single-source benchmark suite for heterogeneous computing,” in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–1.

[29] Codeplay, *SYCL Guide*, 2020 (accessed July 5, 2020). [Online]. Available: https://developer.codeplay.com/products/computecpp/ce/guides/sycl-guide

[30] *Khronos SYCL Overview*, 2020 (accessed July 5, 2020). [Online]. Available: https://www.Khronos.org/sycl/

[31] I. Rudkin, *SYCL for Safety Practitioners – An Introduction*, 2020 (accessed July 5, 2020). [Online]. Available: https://www.codeplay.com/portal/blogs/2020/07/24/sycl-safety-part-1.html

[32] J. Stephan, “Innovative spracherweiterungen für beschleunigerkarten am beispiel von sycl, hc, hip und cuda: Unterschung zu nutzbarkeit und performance,” *TU Dresden*, 2019.

[33] J. Lawson, M. Goli, D. McBain, D. Soutar, and L. Sugy, “Cross-platform performance portability using highly parametrized SYCL kernels,” *CoRR*, vol. abs/1904.05347, 2019. [Online]. Available: http://arxiv.org/abs/1904.05347

[34] H. C. da Silva, F. Pisani, and E. Borin, “A comparative study of sycl, opencl, and openmp,” in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 2016, pp. 61–66.

[35] W. Shin, K.-H. Yoo, and N. Baek, “Large-scale data computing performance comparisons on sycl heterogeneous parallel processing layer implementations,” *Applied Sciences*, vol. 10, no. 5, p. 1656, 2020.

[36] M. Odendahl, *Mastering Automotive Software Development*, 2020 (accessed July 5, 2020). [Online]. Available: https://www.silexica.com/blog/ros-automotive-software-development/

**Markus Glaser** –
SW Project Lead Mercedes Benz
markus.glaser@daimler.com

**Charles Macfarlane** –
VP Marketing at Codeplay
charles.macfarlane@codeplay.com

**Benjamin May** –
CEO AMX13
benjamin.may@amx13.com

**Dr. Sven Fleck** –
CEO SmartSurv
fleck@smartsurv.de

**Lukas Sommer** –
Research Associate at Embedded Systems & Applications Group (ESA), TU Darmstadt
sommer@esa.tu-darmstadt.de

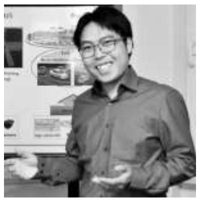
**Illya Rudkin** –
Principal Software Engineer at Codeplay
illya.rudkin@codeplay.com

**Jann-Eve Stavesand** –
Head Of Consulting at dSPACE
jstavesand@dspace.de

**Dr. Stefan Milz** –
Head of R & D – Managing Director Spleenlab.ai
stefan.milz@spleenlab.ai

**Christian Weber** –
Head of Advanced Engineering ADAS at Continental
christian.10.weber@continental-corporation.com

**Rainer Oder** –
CEO AOX Technologies
rainer.oder@aox-tech.de



**Dr. Duong-Van Nguyen** –
Head of ADAS at Panasonic Automotive
Duongvan.Nguyen@eu.panasonic.com

**Frank Böhm** –
CEO hepaclouds
frank.boehm@heapclouds.com

**Enda Ward** –
Camera Architecture & Technology Group Leader / Valeo Master Expert
Valeo Detection Vision Systems (DVS)
enda.ward@valeo.com

**Benedikt Schonlau** –
Manager Consulting4Drive
b.schonlau@consulting4drive.com

**Oliver Hupfeld** –
CEO Inno-Tec Innovative Technology
ohupfeld@in-technology.eu

**Prof. Dr-Ing. Andreas Koch** –
Technische Universität Darmstadt
Computer Science Department (FB20)
Embedded Systems & Applications Group (ESA)
koch@esa.tu-darmstadt.de