

Enabling Unit Testing of Already-Integrated AI Software Systems: The Case of Apollo for Autonomous Driving

Miguel Alcon
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
miguel.alcon@bsc.es

Hamid Tabani, Jaume Abella, Francisco J. Cazorla
Barcelona Supercomputing Center
Barcelona, Spain
hamid.tabani@bsc.es, jaume.abella@bsc.es, francisco.cazorla@bsc.es

Abstract—The advanced AI-based software used for autonomous driving comprises multiple highly-coupled modules that are data and control dependent. Deploying those already-integrated software frameworks makes unit testing, a fundamental step in the validation process of critical software, very challenging in safety-critical systems. To tackle this issue, in this paper, we show the steps we followed to develop standalone versions of the modules in an industry-level autonomous driving framework (Apollo) by applying several modifications to its architectural design. We show how the standalone modules have the same functional behavior as their integrated counterpart modules. We exemplify the benefits of standalone modules by performing incremental analysis of the software timing requirements of each module running on a heterogeneous System on Chip (SoC). This is a mandatory step to consolidate and integrate software modules guaranteeing timing constraints (e.g. related to freedom from interference) while maximizing SoC utilization.

Index Terms—Unit testing, autonomous driving, Apollo

I. INTRODUCTION

The amount and complexity of the software is on the rise in every embedded product in safety-related domains like automotive and avionics [1]–[3]. Software, already acknowledged as one of the most important elements to increase the competitiveness edge of embedded products, increasingly controls more on-board functionalities. To that end, software handles big amounts of data and comprises Artificial Intelligence (AI) algorithms for the realization of several functionalities. For instance, Autonomous Driving (AD) in cars builds on deep learning for object detection and tracking, path planning, driver-monitoring systems, and voice-based command and control [4]. AI is also projected to have a major impact in avionics and space, making systems more autonomous, possibly requiring no human intervention to operate [5], [6].

Leading AI companies provide several machine learning frameworks as well as underlying optimized libraries to exploit performance in available platforms, which are used in diverse domains from improved content discovery to voice/image recognition [7]. However, generic AI algorithms (software) are designed according to no safety standard [8]. This is a conundrum for OEMs and TIER companies in automotive

and avionics: while they want to get access to the increased-precision available AI algorithms (e.g. in object detection), the validation of that software in adherence to applicable safety standards can be costly [9]. This calls for a middle-ground solution to get the best of both worlds.

Safety-related functionalities in embedded systems require domain-specific design, verification and validation (V&V) means that help to the systematic identification of implementation errors. Adhering to existing guidelines (e.g. ISO 26262 and ISO/PAS 21448 for road vehicles) during the development cycle makes the software correct by design. That is, the architecture of the software is specifically designed to meet all safety requirements in an efficient manner so that residual risk of a functional misbehavior of the software is negligible¹.

Also, the design and implementation of the system in general, and software in particular, follows specific practices such as assessing input value properties and other conditions, adhering to MISRA C coding practices, and the like. Following those practices favors code simplicity, modularity, clarity and correctness in general, which in turn eases software verification methods (e.g. formal verification of the functionality) as well as validation processes (i.e. testing).

Software testing is instrumental for AI-based frameworks as they not match well the nature of the usual safety-related development processes and need specific safety standards like ISO/PAS 21448 (SOTIF) for automotive. This is due to the difficulties to specify, design and verify AI frameworks that rely much more on testing processes. The challenge lies in generating a limited number of test scenarios providing sufficient coverage for both, low-level properties of the software (e.g. code coverage, boundary values, timeliness) and high-level properties (e.g. driving scenarios for AD frameworks). The testing process starts at unit level assessing and quantifying low-level properties so that subsequent integration stages can focus on testing integration interfaces and higher-level properties.

¹The residual risk is the amount of risk remaining after specific risk mitigation measures are put in place.

When it comes to software timing, validating each software unit allows narrowing down potential timing issues (e.g. due to errors in the design and verification process that might cause issues with timeliness of real-time software), as well as assessing the impact of integration on timing, which relates to the needed *freedom from interference* (see ISO26262-6 Annex D). Hence, in case of overruns it is easier to single out the cause of the overrun, whether it is caused due to a specific software unit, or due to the impact of integration of several of them.

These principles confront with AD software, as a representative of AI-based software, that encompasses different *already-integrated* highly-coupled modules, each with high cyclomatic complexity [9]. Hence, the key question is how to facilitate unit testing for software whose units (e.g. runnables, tasks and software components) are just deployed as a tightly-coupled framework for efficiency reasons. We help to reduce this gap by illustrating the process we followed to split Apollo AD software framework into its components (units) so that they can be tested in isolation. To that end, our contributions are:

- We analyze the internal software structure of Apollo modules and identify the mechanisms used for communication among modules as well as the corresponding interfaces (Section III). We further discuss about other safety-relevant AI-based frameworks (e.g. Autoware [10]), and show that they have similar properties to those of Apollo on which our approach builds, hence illustrating that analogous processes can be followed for those other frameworks.
- We illustrate the process we followed to generate standalone modules (Section III). We also assess that each standalone module performs the same exact functionality following the same execution pattern that baseline integrated Apollo modules (Section IV).
- We illustrate the benefits of using standalone modules by focusing on a specific verification step: software timing analysis (Section V). We show how standalone modules allow performing an incremental analysis of the software timing requirements of each module when run in a multi-core processor. We are able to set apart the contention that modules generate to each other as we incrementally integrate. We also assess the impact of different deployment scenarios of Apollo when the target system has 4 cameras, not just one as in the baseline case. Finally, we evaluate the sensitivity of each module to the contention generated by a different application that can be integrated into the same platform as Apollo to increase resource utilization. This form of module robust software testing is required to consolidate and integrate modules to maximize processor utilization while meeting freedom from (time) interference requirements and guaranteeing that performance is sufficient, therefore adhering to ISO 26262.

In the final segment of the paper, Section VI discusses the

related work and Section VII presents the main conclusions of this work.

II. BACKGROUND AND MOTIVATION

The software development processes for safety systems include specification, design and verification prior (and concurrently) to software coding. After implementation, those steps are followed by validation (testing) as software is incrementally integrated with other software and hardware components, building subsystems up to the main system [11].

A. Benefits of Unit Testing for Software Timing and Beyond

Validation tests, which start with unit testing, (i) help developing quantitative evidence on the ‘correctness’ of the derived software timing (worst case execution time) estimates and whether those estimates remain valid upon integration, which relates to *freedom from interference* requirements in ISO26262 for software. Those tests also (ii) allow assessing the resource usage of each individual unit like memory and stack usage. Hence, software unit testing contributes to validating upperbounds to required resources (ISO26262-6 clause 7.4.17).

The validation of each individual software unit allows detecting and addressing timing issues such as errors in the production of budgets for the duration of software during the verification process. Besides, having per-unit execution times also allows addressing the impact of incremental integration of the different software units. As the amount of software increases, and so it does the amount of software integrated in each multicore process, singling out the cause of a timing violation (overrun) is instrumental to reduce software development costs.

In addition to software timing, enabling unit testing provides much higher controllability, facilitating reaching high statement, branch and modified condition/decision coverage (MC/DC) (see ISO26262-6, clauses 9.4.4 and 9.4.5), performing resource usage tests, like memory usage (see ISO26262-6 clause 9.4.3), and performing interface testing (see ISO26262-6, clauses 10.4.3 and 10.4.4) by controlling the test cases used for each software unit.

Beyond certification, isolating software units allows reusing software components for their integration in different systems without replicating design and V&V costs for the software component (e.g. integrating the perception module in an autonomous car with different number of sensors, or with improved components for other stages of the AD software).

B. Challenges brought by AD software

1) *Unit Testing*: Unit Testing is the first step and it builds on the following three activities.

Test case generation aims at achieving high coverage of potential software failures with the lowest test effort possible, which generally implies minimizing the number of test cases. Equivalence classes are defined for tests, where software is analyzed considering potential inputs to determine groups of

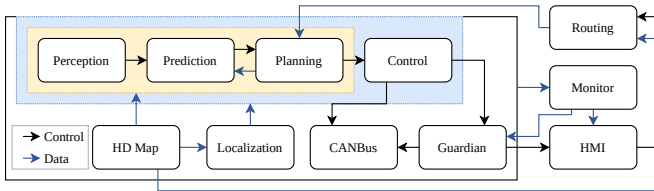


Fig. 1. Apollo modules and control/data dependencies.

inputs (i.e. an equivalence class) that test the same statements, branches and conditions, so that only one or few of those tests in one class are used for testing. Other tests, despite redundant with equivalence classes, can be added to test boundary values for inputs and cases that may lead to errors based on engineering expertise.

AD software challenges: Test generation builds on *software controllability* to enforce specific inputs for each software unit. The lack of controllability (e.g. in an integrated framework) makes it challenging generating system tests leading to specific unit inputs, which usually translates into the inability to test some equivalence classes and the over-testing of some others.

Definition of coverage metrics. Coverage metrics like statement and MC/DC define how test coverage needs to be quantified. *Observability* of the software unit is needed to quantify such coverage metrics (e.g. profiling code).

AD software challenges: The inability to test software units in isolation may lead to overly large traces for all the software framework at once, or long tests to collect traces for each unit by running the whole framework.

Test execution and result assessment. Tests are checked for correctness by comparing obtained results against expected results and/or expected properties for the results (e.g. values within specific bounds). Coverage metrics and resource usage information (e.g. execution time, memory used) are also collected for software units.

AD software challenges: Assessing correctness, coverage and resource usage for a software unit tested in isolation is generally easy, since each unit execution delivers a set of results corresponding strictly to the unit. Extracting per-unit information from a whole framework test battery is, instead, more laborious and prone to errors – if at all possible.

2) *Integration Testing:* After testing software units individually, they are integrated incrementally among them and with hardware components like sensors and actuators.

AD software challenges: Per-unit tests are no longer performed and, instead, integration tests are performed such as those intended to test function and call coverage, unit interface testing, and resource usage testing at a coarser granularity.

3) *Conclusions:* Overall, both unit and integration testing heavily build on controllability and observability at the granularity at which testing is performed. Hence, fully integrated frameworks challenge unit testing and, potentially, integration testing if it needs to be conducted at a finer granularity than the whole framework.

III. SPLITTING INTO STANDALONE MODULES

A. Apollo Autonomous Driving Software

The Apollo [12] open-source AD software framework is provided by Baidu. It has 120+ industrial partners, most of which are top-TIER AI companies and car manufacturers. Apollo has been deployed on a variety of prototype vehicles (including autonomous trucks) and supports state-of-the-art hardware such as the latest LiDARs and cameras, from different vendors, as well as GPU acceleration. In this work, we focus on Apollo version 3.

Apollo modules and even functions within modules are highly coupled (e.g., no prediction of trajectories is possible if objects are not perceived in an autonomous vehicle). Each module runs as an executable. The execution is organized into stages, each of which is allocated to a specific functional step in each module. Hence, all executable modules run concurrently on a recurring basis. Figure 1 shows Apollo modules as well as their dependencies. The main Apollo modules are: (1) *CANBus* is the interface that passes control commands to the vehicle hardware and chassis information to the software system. (2) *Control* executes the planned spatio-temporal trajectory by generating control commands such as accelerate, brake, and steering. (3) *Guardian* is a safety module that performs the function of an Action Center and intervenes should monitor detect a failure. (4) *HD Map* provides ad-hoc structured information of the roads. It is used as a library, working as query engine support that is used by Perception, Prediction, and Planning, and Routing. (5) *HMI* (Human Machine Interface) or DreamView in Apollo is a module for viewing the status of the vehicle, testing other modules and controlling the functioning of the vehicle in real-time. (6) *Localization* estimates where the vehicle is located with centimeter-level accuracy, using various information sources such as GPS, LiDAR and Inertial Measurement Unit (IMU). As shown, Localization provides data to Perception, Prediction, Planning and Control modules. (7) *Monitor* is the surveillance system of all the modules in the vehicle including hardware. (8) *Perception* identifies the area surrounding the vehicle by detecting objects, obstacles, and traffic signs. It is considered the most critical and sophisticated module of an AD system. Perception also fuses the output of several types of sensors such as LiDAR, radar, and camera to improve its accuracy. (9) *Planning* plans the spatio-temporal trajectory for the autonomous vehicle to take. (10) *Prediction* anticipates the future motion trajectories of the perceived obstacles. (11) *Routing* tells the autonomous vehicle how to reach its destination via a series of lanes or roads.

B. Apollo Software Architecture

Apollo modules share a similar execution pattern starting with the main function that performs the initialization of some libraries and parsing of the configuration files and command-line parameters. One of these libraries is the Robot Operating System (ROS), which is a set of software libraries and tools

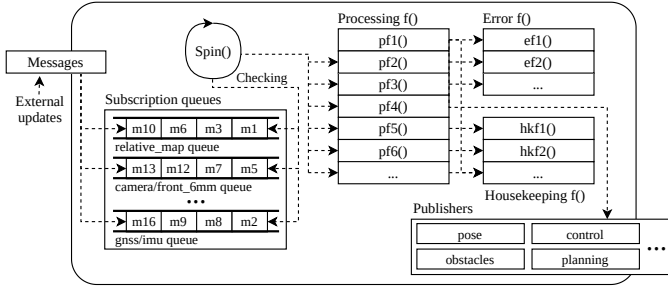


Fig. 2. Internal operation of an Apollo module.

that help to build robot applications and is widely used in autonomous driving [10], [12].

A module is *subscribed* to a topic (a type of message) if it reacts to it by invoking a function (*callback*) defined within the module. A module can have several callbacks for the same topic. All modules use adapters to communicate with I/O (ROS) and with other modules, and each adapter is attached to a certain topic. A module can *publish*, i.e. broadcast, messages of a certain type.

At initialization, a spin function is called (see Figure 2) to initialize the modules, and create and start a ROS *spinner*. Depending on the parameters, it can become a single or multi-threaded spinner. The spinner is in charge of looking for input messages and processing them until the user shuts down the module. Given a message, a spinner is responsible for identifying all the callback functions related to it, as long as the module is subscribed to the message’s *topic*. Processing functions, $pf_i()$ or callbacks, in a module are called by the spinner when they are subscribed to the topic’s incoming message and hence, they process those types of messages. As an example, when the front camera publishes an image with the topic ‘front camera’, the spinner calls all the processing functions that are subscribed to this topic. Each of these functions may process the message in different ways. For instance, the same image can be used for both detection (obstacles, lane lines, etc.) and recognition (signals, traffic lights, etc.) purposes, so the module may handle them with different functions. Error functions, $ef_i()$, verify the functional correctness of the processing functions, from which they are usually called functions. Housekeeping functions, $hkf_i()$, perform the required processes before the spinner starts to look for incoming messages. This ranges from the initialization of the libraries like ROS, to the parsing of the module’s configuration files or command line parameters.

After initialization, the module enters into a steady state, in which the module performs operations on messages that arrive. If a message enters the system, via ROS or another module, the adapter (with the ROS callback function) will trigger all functions inside the vector for the message’s topic. Note that the spinner created by the module is the one detecting the message, and the one running the function.

C. Module Splitting

Our approach to develop standalone modules consists of two main stages: (1) stripping out the data to feed the standalone

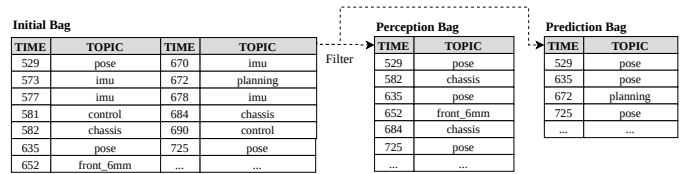


Fig. 3. Modifications to split the content of bag files.

modules and (2) making the modules fully standalone so that, by building solely on external data, they neither are susceptible to the behavior of other modules running concurrently (including other instances of their own), nor alter the behavior of other modules. For the first stage, we propose to create module-specific datasets from the global dataset, (*bag splitting*), which we extend with data from the integrated version (*data generation and timestamp adjustment*) that is necessary for the correct execution of the modules. For the second stage, we propose to force all modules to process all the incoming data (*lifespan adjustment*), and we disable messages publication (*disable publications*) to ensure that the only data the module is processing comes from external sources, thus avoiding interference when running more than one standalone module at the same time.

Bag splitting. When running AD frameworks on a machine that is not within a car, the user needs external data to simulate the information a car would receive in real scenarios. Several AD systems such as Apollo rely on ROS “bags”, a file format used by ROS-based programs for storing messages data that can contain any type of information. In Apollo, messages are used for communication between modules and with the various sensors. Each message is attached to a certain topic (e.g., front camera, LiDAR sensor, output of perception, etc.).

Running Apollo modules implies running all its modules concurrently. Thanks to ROS *bags*, a single module can also run independently if a *bag* is played back (i.e. all the messages inside the bag will be sent to the system just as they were recorded, in the same order and lapse of time) and it contains all the data that the module requires. Hence, a module processes messages from the sensors or other modules that were recorded in a real scenario. However, for some modules not all the data it needs may be included in *bags*, and instead be only produced by other modules when run concurrently.

Bag files provided along with Apollo are sufficient to run the integrated framework, but not an individual module since those bags lack some messages produced by some modules when running the whole framework. However, running the complete framework with the original bag files triggers other modules, thus impeding unit testing. For this reason, we generated one *bag* per module only with the respective data they require. To do so, we developed a script, using the *roscpp* API [13], that generates a ROS *bag* from another one, taking only the messages a given module needs. With the new produced bags, we are able to run modules in isolation without any source of interference from the rest of modules. Figure 3 shows how a generic initial bag file is split into multiple bag files (one

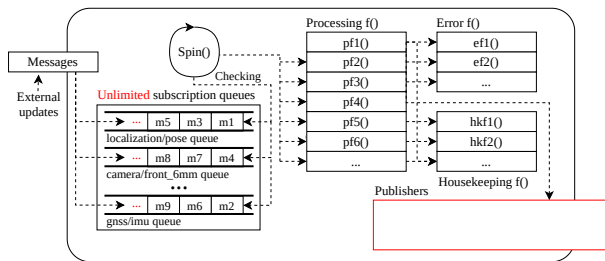


Fig. 4. Modified internal operation of an Apollo module.

per module) that contain only messages for the corresponding module. This step requires identifying each of the topics that a module is subscribed to and adjust their timing.

Data generation and timestamp adjustment. To make ROS *bags* complete, we accurately tracked the data that modules generate during their normal execution, dumped it and integrated it into the split bag files produced to make each module run independently with the complete set of data they would use if run as part of the integrated framework. Furthermore, timestamps of each message in the *bag* files were adjusted to allow running the modules in isolation. This is needed to guarantee that modules are not triggered earlier than allowed when increasingly integrated with other modules.

Lifespan adjustment. When we compare the number of messages processed and the callback functions launched in the original version and our standalone version, we realize that Apollo can limit the number of messages each topic can store (not all messages can be processed as soon as they arrive). This can lead to the drop of messages that are too old. This is done due to the fact that each message has a lifespan and, once it is not used during its lifespan, the message becomes useless. However, we modified Apollo to move this limit far enough, so we let it process all messages that are inside the bag without dropping any of them, thus gaining determinism for testing, as shown in Figure 4. This is essential in order to compare different executions of Apollo by having the certainty that all of them will process *exactly* the same messages, so they are functionally identical. Nevertheless, in a real scenario, this lifespan adjustment may not be required, since old messages become irrelevant after a certain time period.

Disable publications. To maintain the same behavior in all executions, we disabled the publication of messages, as shown in Figure 4. With this, we ensure that all modules only process the messages inside the bag, and do not generate redundant messages. Otherwise, the module under analysis would likely produce and publish one output that could be already in the bag, i.e. the same message was also generated during the recording of the bag, so that two identical messages would be received by the subscribed modules, which would not happen in a real scenario. It should be mentioned that disabling message publication is only done for the sake of avoiding functional interference across different modules executing simultaneously since standalone modules in isolation are not functionally affected by their own messages.

D. Other AD frameworks

Our analysis of several autonomous driving systems shows that they follow a similar software architecture. For instance, Autoware [10], [14], analogously to Apollo, builds upon ROS. ROS determines the way the different modules interface each other by means of callbacks, which are used to subscribe to specific messages that are sent by source modules. Modules in both, Autoware and Apollo, exchange specific messages and have similar structures across both frameworks, thus building on input sensors that provide data periodically (e.g. cameras, LiDAR, radar), and modules processing such data (perception, prediction, planning, etc.). Performance demanding functionalities, such as those in charge of object detection, build upon open libraries for machine learning, which ultimately use neural networks that run on GPUs for both frameworks. Other frameworks such as NVIDIA Drive [15] use analogous schemes to connect modules and to implement some functionalities such as object detection and 3D distance prediction with neural networks [16], [17].

IV. FUNCTIONAL VALIDATION

We target an x86 architecture with an NVIDIA 1080 Ti GPU. It comprises an octa-core AMD Ryzen 7 1800X processor in which each core is dual-threaded. Each core has private 32 KB data and 64 KB instruction caches, and private 512 KB inclusive L2 caches. The 2 x 8 MB exclusive L3 caches are shared among all cores. The discrete GPU has 3584 CUDA cores and 11 GB GDDR5X dedicated memory. The automotive industry relies for some products on the use of GPUs with discrete memory such as NVIDIA Drive Pegasus [15].

We run Apollo inside docker images similar to its configuration on real vehicles and the entire framework is run on top of a Linux operating system. High-performance GPUs employ high-bandwidth discrete memory to provide the required performance for memory-intensive workloads such as object detection or other deep learning workloads. We use representative data sets and input videos provided by Apollo to run the modules and perform our experiments [12].

We developed several scripts to execute Apollo modules while running the ROS bags always in the same manner (i.e. deterministically) and to parse the output of Apollo to collect the results we need. To collect GPU kernels information, we used the NVIDIA profiler tool, *nvprof* [18]. We focus on callback (processing, pf_i) functions as they are the ones that are triggered during the steady operation of Apollo. Moreover, these functions usually take care of the core functionalities of modules, making them the target of our experiments. We focus on six representative modules (the rest either bring no additional insights or can only be instantiated in the real car hardware): Perception, Prediction, Planning, Localization, Guardian and Control. We specially focus on Perception for space constraints as it is the most time-consuming module in Apollo. The results for the other modules are summarized in Table I.

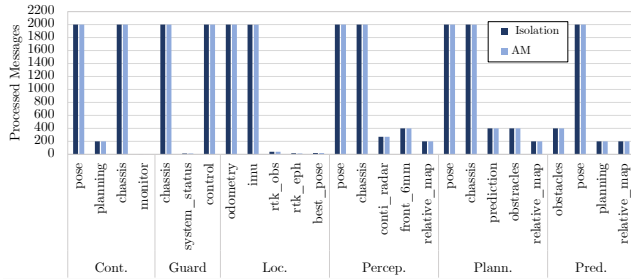


Fig. 5. Number of messages processed in Apollo by standalone and integrated modules. The X-axis shows the type of message (topic).

Figure 5 shows the number of processed messages by the analyzed Apollo modules, for both our standalone version and the integrated (default) version modules. As it can be seen, the number of messages of each type processed is identical. Besides, we positively assessed that the function call sequences in both cases match. We also performed several tests using various data sets to make sure that, in both cases, the module produces the same output as in the integrated baseline.

V. INCREMENTAL/ROBUST TIMING TESTING

We cover the following incremental integration and robust testing scenarios.

A. Incremental Integration

As software size increases in every new product, software modules and drivers of an AD system are designed, developed and implemented by different providers. Usually, during early stages the final configuration of a system in which the module will be integrated may not be fully known to providers. For instance, the Perception module may be integrated in different deployments with different sensor types (camera, LiDAR, and/or radar), as well as with a varying number and type of cameras (e.g., long-range, wide, etc.) and used resolution. It is then key to consider multiple software integrations.

In order to capture this scenario, we conduct a series of experiments building on our standalone Apollo modules in which we incrementally integrate them to assess the impact that each integrated module causes on the rest. Figure 6 shows the execution time variability of the main `ImgCallback` function (which is the one that processes the images) in Perception as other Apollo modules are incrementally integrated: Prediction, Localization, Planning, Control, Routing, CAN bus, Monitor and Guardian. The variability in each integration scenario is shown with a whisker plot capturing min, first quartile, median, third quartile, max, and outliers. As shown, our standalone modules allow assessing that the impact of contention is small and within the range of the standard deviation, thus not being statistically significant. This occurs because Perception execution time is dominated by the GPU, where deep learning object detection runs, and the other modules do not use the GPU. Moreover, the discrete GPU uses its own memory interfaces (i.e. not shared with the CPU). When analyzing the other modules and their main functions (see medians in Table I), we also conclude that

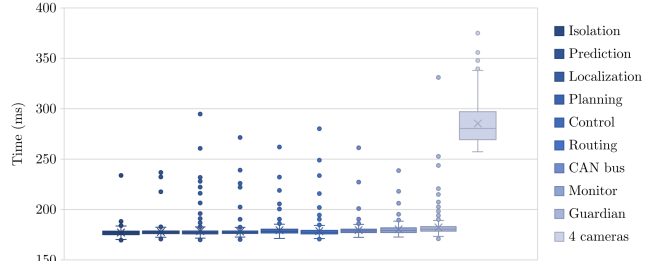


Fig. 6. Perception `ImgCallback` timing as more modules are integrated

variability is not statistically significant due to the fact that they mostly use core-local resources, thus being highly insensitive to interference. Note that this analysis would not be possible with the already-integrated framework.

B. Different deployment scenarios

We also consider another integration scenario with 4 cameras instead of 1 (the default case). This covers the integration scenario in which the autonomous vehicle can have multiple cameras, to cover different angles and/or ranges [19]. In this case, all the processes related to processing camera inputs, such as deep learning workloads for perception, are quadrupled. We implemented this configuration by adding each extra camera process as a new callback to the system. Note that in the 4-camera scenario the output produced by each camera is processed by a different callback function. Hence, the computations made by every callback function is the same as in the 1-camera scenario. As Figure 6 shows, the slowdown due to contention in the access to multi-core processors is as high as 60% when three callbacks are added to the system in the 4-camera scenario.

C. External Applications

The reduced margins in automotive calls for increasing resource utilization to reduce costs. Electronics is not an exception to that, so it is required the safe consolidation of more applications onto the same SoC. However, this can cause each application to suffer a higher slowdown due to contention in the access to shared resources.

In order to capture this scenario, we developed cache-aggressive CPU benchmarks that access a large data array with either writes or read/writes that trash the L3 and go to memory. They are called `CPU_st` and `CPU_ldst` respectively. Also, as some modules are accelerated using the GPU, we also developed GPU benchmarks that deploy different number of threads to be run concurrently with the GPU kernels of the Apollo module(s). GPU benchmarks are developed as streams to be added to the main kernel and their concurrent execution is verified using profiling visualization tools.

Each module deploys a different number of software threads that are mapped to the hardware threads and cores (recall cores implement hyper-threading). In the worst case, we have observed that perception uses 6 threads, hence leaving 10 hardware threads available and unused. For this reason we run up to 10 copies of `CPU_ldst` and `CPU_st` benchmarks in

TABLE I
NORMALIZED MEDIAN OF THE EXECUTION TIME OF ALL CALLBACK FUNCTIONS WITH RESPECT TO THE ISOLATION SCENARIO.

Module	Callback	ISO	AM	4CAM	ISO+CPU_st	ISO+CPU_ldst	ISO+GPU_1kt	ISO+GPU_2kt	ISO+GPU_4kt	ISO+GPU_8kt
Control	OnMonitor	1	1	-	1	4	-	-	-	-
	OnTimer	690	737	-	900	1900	-	-	-	-
Guardian	OnChassis	6	5	-	28	9	-	-	-	-
	OnControl	8	8	-	53	18	-	-	-	-
	OnSystemStatus	84	82	-	177	101	-	-	-	-
	OnTimer	65	47	-	434	183	-	-	-	-
Localization	OnRawImu	62	63	-	95	211	-	-	-	-
Perception	ImgCallback	176892	180525	280432	238469	237692	452712	469637	539435	744039.5
	OnChassis	6	8	8	29	29	7	7	9	8
	ImageCallback	1	1	0	3	2	1	1	0	0
	OnLocalization1	62	84	85	276	259	75	80	98	101
	OnLocalization2	15	16	16	59	57	15	15	18	15
	OnRadar	5530	5539	3904	7043	6849	5430	5312	4599	3888.5
Planning	OnTimer	1096	1157	-	1977	3745	-	-	-	-
Prediction	OnLocalization	24	24	-	37	83	-	-	-	-
	OnPlanning	210.5	209	-	264.5	574.5	-	-	-	-
	RunOnce	784.5	762.5	-	1102	2127	-	-	-	-

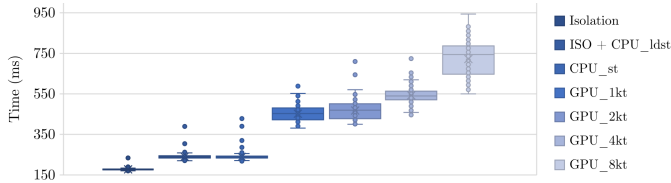


Fig. 7. Perception ImgCallback timing with benchmarks

order to increase resource utilization to 100%, with Figure 7 reporting results for 10 CPU threads. In particular, the second and third bars of Figure 7 show the slowdown when the standalone Perception ImgCallback runs against CPU benchmarks: *CPU_ldst* and *CPU_st* result in a (median) slowdown of 34% and 35% respectively.

Regarding GPU benchmarks (last 4 bars in Figure 7) we consider two scenarios. When the number of threads is below the GPU capacity, GPU_1kt (1kt = 1024 threads) and GPU_2kt in Figure 7, we can run the benchmark completely in parallel with the Perception module. We have confirmed this using GPU profilers and visualization tools and concluded that the observed slowdown (2.56x and 2.65x in the median respectively for 1kt and 2kt) is due to contention only. However, when the number of threads exceeds the GPU capacity, i.e. GPU_4kt and GPU_8kt in Figure 7, the CUDA runtime time-shares the GPU between Perception and the benchmark creating another source of delay slowing down Perception ImgCallback by up to 4.21x.

Table I summarizes the observed impact on the callback functions of all analyzed modules of the CPU and GPU benchmarks. We see a variety of effects of contention across different modules with functions more/less sensitive to adding additional cameras, CPU load, and GPU load. The case of OnRadar in the Perception module is interesting as it decreases its execution time when in the 4 camera setup. Our initial analysis suggests that this function, is particularly sensitive to CPU contention. Thus, when adding more cameras, or GPU_8kt, the execution time increases due to serialization in the GPU, and hence, it is more likely that OnRadar executes with fewer simultaneous contenders in the CPU. Instead, when running concurrently with *CPU_st* or *CPU_ldst*, its execution time

increases noticeably. Again, note that the ability to perform this type of analysis in a per-module basis, with the appropriate degree of observability and controllability is only possible with our standalone modules. Overall, standalone modules allow to carry out unit testing for multiple deployment scenarios easily, thus gaining confidence on the validation of the system that would not be achieved otherwise.

VI. RELATED WORK

Several works assess functional safety for AD systems. Prior to the publication of the SOTIF standard, where specific safety considerations are made for AD systems, Gosavi et al. [20] analyzed the software architecture of AD systems and showed that ISO 26262 is not enough to prove their functional safety. Helmut et al. [21] also reviewed issues concerning the availability, reliability, and architecture of AD systems, and proposed ways to handle these challenges and to adapt ISO 26262. Similarly, after SOTIF publication, Shuler [22] concludes that ISO 26262 is not enough for AD systems, and instead, SOTIF is the appropriate standard extension to cover this gap but regards SOTIF as “too philosophical to be actionable in engineering design and validation practices”. Overall, those works analyze aspects related to the architecture of the system prior to its implementation, and do not consider practical implementation and testing concerns. Tabani et al. [8] consider implementation concerns and analyze and quantify elements such as coding guidelines, code coverage, and software complexity. However, how to enable practical testing of already-integrated AD frameworks during validation phases has not been addressed so far. Our work aims at covering this gap.

In the context of unit testing, industry has resorted so far to minor extensions of single-core methodologies to address multi-core challenges [23], [24] building on the fact that either (1) multi-cores used are automotive devices such as, for instance, the Infineon AURIX family, with low core counts (e.g. 3 to 6) and high degrees of controllability and deterministic timing by design, or (2) high-performance multi-cores and accelerators (e.g. GPUs) are used as QM devices for fail-safe applications, thus decomposing the system into an ASIL

automotive multi-core and a QM high-performance device, so that failures in the latter may lead only to availability concerns, but not safety concerns. However, the advent of high autonomy levels for AD systems leads to fail-operational AD platforms, thus imposing some ASIL on AD frameworks, which therefore need to be proven compliant against their safety requirements in accordance with safety regulations (i.e. ISO 26262 and SOTIF). However, solutions for already-integrated software frameworks, such as the one presented in this paper, have not been devised before.

Timing analysis of AD systems proved to be challenging and complex. Authors in [25] show the main challenges and limitations in finding a satisfactory software timing analysis solution for Apollo. All the analysis in this work is based on the entire Apollo system with all modules running concurrently. Pujol et al. [26] and Liu et al [27] show how deep learning features are integrated in the latest autonomous driving frameworks and bring challenges related to scheduling and the use of sophisticated heterogeneous SoCs. These works also consider already integrated modules, unlike this work where we analyze standalone modules.

VII. CONCLUSIONS

Unit testing is fundamental in automotive development processes for safety-related software as it allows assessing the correctness of individual software components (e.g. functions, tasks) with appropriate controllability and observability levels. However, in the context of AD, automotive industry embraces already-integrated frameworks as a means to inherit the most advanced AI-based technologies for object detection and tracking. This challenges testing since units (e.g. their modules) cannot be tested independently, bringing uncertainty on the safety of the AD system once deployed. We tackle this challenge by providing a practical solution to decompose AI-based AD integrated frameworks into standalone modules easing their validation by enabling unit testing. For Apollo we show functional equivalence between the standalone modules and their integrated counterparts. We also illustrate how those modules can be used to assess the impact of timing interference in MPSoCs in different integration scenarios, including (1) the incremental integration of modules, (2) multiple deployments with different number of sensors (e.g. multiple cameras), and (3) integration with software from other systems or providers with arbitrary impact in timing interference, which we evaluate with stressing benchmarks.

ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Science, Innovation and Universities under grant PID2019-110854RB-I00/AEI/10.13039/501100011033, the UP2DATE European Union's Horizon 2020 (H2020) research and innovation programme under grant agreement No 871465, the SuPerCom European Research Council (ERC) project under the European Union's Horizon 2020 research

and innovation programme (grant agreement No. 772773), and the HiPEAC Network of Excellence.

REFERENCES

- [1] D. Mohr et al., "The road to 2020 and beyond: What's driving the global automotive industry," 2013.
- [2] Ford, "Media Center Release," <https://media.ford.com/content/fordmedia/fna/us/en/news/2017/02/10/ford-invests-in-argo-ai-new-artificial-intelligence-company.html>, 2017.
- [3] Toyota Motor Corporation, "Toyota News Release," <https://corporatenews.pressroom.toyota.com/releases/toyota+establish+artificial+intelligence+research+development+company.htm>, 2015.
- [4] "Self-driving Safety Report," 2018. [Online]. Available: <https://www.nvidia.com/en-us/self-driving-cars/safety-report/>
- [5] "The Future of AI/ML in Avionics," 2018. [Online]. Available: https://blogs.windriver.com/wind_river_blog/2019/01/the-future-of-aiml-in-avionics/
- [6] O. Notebaert, "On-Board Payload Data Processing requirements and technology trends," in *European Workshop on On-Board Data Processing (OBDDP2019)*, 2009.
- [7] M. Abadi et al., "Tensorflow: A system for large-scale machine learning," in *OSDI*, vol. 16, 2016, pp. 265–283.
- [8] H. Tabani et al., "Assessing the adherence of an industrial autonomous driving framework to ISO 26262 software guidelines," in *DAC*, 2019.
- [9] S. Alcaide et al., "Safety-related challenges and opportunities for gpus in the automotive domain," *IEEE Micro*, vol. 38, no. 6, 2018.
- [10] The Autoware Foundation, "Autoware. An open autonomous driving platform." <https://github.com/CPFL/Autoware/>, 2016.
- [11] International Organization for Standardization, *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
- [12] Baidu, "Apollo, an open autonomous driving platform." <http://apollo.auto/>, 2018.
- [13] ROS Wiki. (2020) Rosbag cookbook. [Online]. Available: <http://wiki.ros.org/rosbag/Cookbook>
- [14] S. Kato et al., "Autoware on board: Enabling autonomous vehicles with embedded systems," in *ICCPSS*, 2018.
- [15] "NVIDIA DRIVE - Autonomous Vehicle Development Platforms." 2020. [Online]. Available: <https://developer.nvidia.com/drive/>
- [16] "Deploying a Scalable Object Detection Inference Pipeline." 2020. [Online]. Available: <https://developer.nvidia.com/blog/deploying-a-scalable-object-detection-inference-pipeline/>
- [17] "Deep neural networks trained on radar and lidar data predict 3D distances from 2D images." 2019. [Online]. Available: <https://blogs.nvidia.com/blog/2019/06/19/drive-labs-distance-to-object-detection/>
- [18] NVIDIA. (2019) Profiler :: CUDA toolkit documentation. [Online]. Available: <https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview>
- [19] Tesla Corporation, "Tesla Autopilot," 2018. [Online]. Available: <https://www.tesla.com/autopilot>
- [20] M. A. Gosavi et al., "Application of functional safety in autonomous vehicles using iso 26262 standard: A survey," in *SoutheastCon*, 2018.
- [21] H. Helmut et al., *Functional Safety of Automated Driving Systems: Does ISO 26262 Meet the Challenges?*, 2017.
- [22] K. Shuler, "Taking Self-Driving Safety Standards Beyond ISO 26262," 2019. [Online]. Available: <https://semiengineering.com/taking-self-driving-safety-standards-beyond-iso-26262/>
- [23] C.-S. Koong et al., "Automatic testing environment for multi-core embedded software-ateses," *J. Syst. Softw.*, vol. 85, no. 1, p. 4360, Jan. 2012.
- [24] Hitex GmbH, "Multi-core MCA Support with Universal Debug Engine," 2020. [Online]. Available: <https://www.hitex.com/tools-components/development-tools/pls-development-environment/multi-core-mca-support>
- [25] M. Alcon et al., "Timing of autonomous driving software: Problem analysis and prospects for future solutions," in *RTAS*, 2020.
- [26] R. Pujol et al., "Generating and exploiting deep learning variants to increase heterogeneous resource utilization in the nvidia xavier," in *ECRTS*, 2019.
- [27] S. Liu et al., "On removing algorithmic priority inversion from mission-critical machine inference pipelines," in *RTAS*, 2021.